SequenceLayers: Sequence Processing and Streaming Neural Networks Made Easy

RJ Skerry-Ryan, Julian Salazar, Soroosh Mariooryad, David Kao, Daisy Stanton, Eric Battenberg, Matt Shannon, Ron Weiss, Robin Scheibler, Jonas Rothfuss, Tom Bagby *rjryan@google.com*
Google DeepMind

Abstract

We introduce a neural network layer API and library for sequence modeling, designed for easy creation of sequence models that can be executed both layer-by-layer (e.g., teacher-forced training) and step-by-step (e.g., autoregressive sampling). To achieve this, layers have an explicit representation of their *state over time* (e.g., a Transformer KV cache, a convolution buffer, an RNN hidden state), and a `step` method that evolves that state, tested to give identical results to a stateless layer-wise invocation. A declarative API and suite of combinators enables construction of complex streamable models from simple streamable components with strong correctness guarantees. The SequenceLayers contract enables complex models to be immediately streamable, mitigates a wide range of common bugs arising in sequence processing, and can be implemented in any deep learning library. SequenceLayers is beneficial for both streaming and non-streaming applications alike, thanks to its safety guarantees and composable, declarative API. Our current implementations of SequenceLayers (JAX, TensorFlow 2) are available at `https://github.com/google/sequence-layers`.

1 Introduction

Neural network models that perform sequential processing have become central to deep learning, from sequence-to-sequence models for machine translation, text-to-speech, and speech recognition, to real-time variants which perform live translation, dialogue generation, and transcription, to next-token generative models for language (LLMs), media, and beyond. However, sequence processing has some common pitfalls:

- **Batching sequences of unequal length.** One must track and mask invalid timesteps, and verify that all layers are invariant to padding, including pooling, downsampling, and upsampling operations.
- **Causality constraints.** Modern architectures like the Transformer (Vaswani et al., 2017) often have an efficient parallel ("teacher-forced") training code path implemented separately from the autoregressive sampling path used during inference. Both code paths must be implemented in a way that avoids causality violations.
- **Offline vs. streaming inference mismatch.** Offline (parallel) inference is often implemented via masking on e.g. attention weights. Converting this to a streaming setting where masking is implicit can require re-implementation due to considerations like lookahead windows and memory constraints, causing training and inference disparities.
- **Unnecessary coupling of architecture and algorithm.** Lack of abstraction of architectural details typically leads to unnecessary coupling. For example an `"AutoregressiveTransformer"` couples both the model architecture (Transformer) and the algorithm (factoring the joint probability of sequences autoregressively via the chain rule). The details of autoregressive modeling (mapping from previous timepoints to a parameterized distribution, and sampling from that distribution) are

independent of the choice of architecture (e.g., Transformer, RNN, state-space model, ...), yet are often unnecessarily coupled which can hinder experimentation and produce research technical debt.

To improve the experience of developing sequence models, we propose SequenceLayers, a lightweight *layer* library for sequential neural networks. It has **three core features**:

1. **Streamable.** SequenceLayers gives you streaming *for free*, in a production-friendly way. To achieve this, every streamable layer that has dependencies over time implements an explicit state, plus a step method that evolves that state (Section 2.2).
2. **Correct.** SequenceLayers is *correct by default*, making entire classes of bugs impossible, e.g., those due to masking, upsampling, downsampling, causality, and padding-invariance. This comes from enforcing layer vs. step equivalence (Section 2.3) and tying mask information to sequence data everywhere with `Sequence` objects (Section 2.1).
3. **Composable.** SequenceLayers has an easy-to-understand *declarative* API that enforces guarantees, enabling sequence models with concise definitions that read like block diagrams. These compositions are immediately streamable and expose aggregate properties like overall output ratio, latency, receptive field size, and state (Section 2.4).

An example definition of a Transformer layer is shown in Figure 1. Due to the built-in `DotProductSelfAttention` layer's implementation of state (KV cache) and step, plus the automatic state wrapping and unwrapping performed by the `Serial` and `Residual` combinators, the aggregate layer exposes `get_initial_state()` and `step()` methods, allowing a sampling loop to be written with no further work.

1.1 Related Work

We take inspiration for our declarative and composable API from Trax (Mohiuddin et al., 2019), which demonstrates the effectiveness of serial and parallel combinators (Section 2.4). We simplify their approach by removing the data stack, and add the additional capability of streamability.

While existing libraries such as Keras (Chollet et al., 2018), Trax (Mohiuddin et al., 2019) and others promote a declarative and compositional API, to the best of our knowledge no other layer library or framework leverage explicit state in a uniform API in order to enable streaming of compositions of layers.

Unlike other fully featured deep learning frameworks such as T5X (Roberts et al., 2023), HuggingFace Transformers (Wolf et al., 2019), and Lingvo (Shen et al., 2019), we intentionally limit the scope of SequenceLayers to *layers*. We view training objectives, data loading, and optimization as out-of-scope. This keeps SequenceLayers lightweight and adaptable to one's model framework of choice.

SequenceLayers is a design, not an implementation. Although it is implemented for JAX (Frostig et al., 2018) and TensorFlow (Abadi et al., 2016), its design is agnostic to the underlying neural network framework and could be ported to e.g., PyTorch (Paszke et al., 2019) or MLX (Hannun et al., 2023).

2 Design

2.1 The `Sequence` type

SequenceLayers consume and produce `Sequence` objects, which are lightweight, differentiable, pytree[1] dataclasses that pair sequence `values` of shape `[batch, time, ...]` with a boolean `mask` of shape `[batch, time]` indicating valid positions in the batch. The `...` dimensions following the `[batch, time]` dimensions are the **channel shape** of the `Sequence`. A **spec** is simply a `jax.ShapeDtypeStruct` or `tf.TensorSpec` indicating shape, dtype, and sharding information.

[1] `https://docs.jax.dev/en/latest/pytrees.html`

Flax Transformer block

```python
def setup(self) -> None:
  """Create submodules."""
  self.pre_attention_norm = ...
  self.attn = ...
  self.post_attention_norm = ...
  self.attn_dropout = ...
  self.pre_ffw_norm = ...
  self.mlp = ...
  self.post_ffw_norm = ...
  self.ffw_dropout = ...

def __call__(
  self,
  x: Float[ArrayT, 'B L D'],
  input_mask: Bool[ArrayT, 'B L'],
  attn_cache: AttentionKVCache | None,
  training: bool,
) -> tuple[Float[ArrayT, 'B L D'],
           AttentionKVCache | None]:
  # Self attention.
  h = self.pre_attention_norm(x)
  attn, new_cache = self.attn(
    h, attn_cache=attn_cache)
  attn = self.post_attention_norm(attn)
  attn = self.attn_dropout(
    attn, training=training)
  x = x + attn

  # Gated GeLU FFN.
  norm_x = self.pre_ffw_norm(x)
  ffw_x = self.mlp(norm_x, input_mask)
  ffw_x = self.post_ffw_norm(ffw_x)
  ffw_x = self.ffw_dropout(
    ffw_x, training=training)
  output = x + ffw_x
  return output, new_cache

y = block(x, mask)
```

SequenceLayer Transformer block

```python
import sequence_layers.jax as sl

block = sl.Serial.Config([

  # Self attention.
  sl.Residual.Config([
    sl.RMSNormalization.Config(
      name='pre_norm'),
    sl.DotProductSelfAttention.Config(
      num_heads=16,
      units_per_head=64,
      # Global causal attention.
      max_past_horizon=-1,
      max_future_horizon=0,
      name='self_attention'),
    sl.DenseShaped.Config(
      [d_model], name='out_proj'),
    sl.RMSNormalization.Config(
      name='post_norm'),
    sl.Dropout.Config(dropout_rate),
  ], name='attention_block'),

  # Gated GeLU FFN.
  sl.Residual.Config([
    sl.RMSNormalization.Config(
      name='pre_norm'),
    sl.Dense.Config(
      4 * d_model, name='dense1'),
    sl.GatedUnit.Config(jax.nn.gelu, None),
    sl.Dense.Config(d_model, name='dense2'),
    sl.RMSNormalization.Config(
      name='post_norm'),
    sl.Dropout.Config(dropout_rate),
  ], name='ffn_block')

], name='transformer_block').make()

y = block.layer(x, training=False)
```

Figure 1: Imperative forward pass definition of a Transformer block in Flax, versus a declarative definition in SequenceLayers for JAX. Unlike other declarative libraries, this SequenceLayer definition gives you a stateful `step()` method out of the box, due to the contracts implemented by each component layer and combinator. Imperative `setup()` methods and implementation overrides remain available, allowing the mix-and-match of paradigms.

`Sequence`s are array-like objects, in that they implement properties of `jnp.ndarray` or `np.ndarray` like `shape`, `dtype`, and `ndim`. However, `Sequence` does not implement the array protocol[2] to avoid accidentally using a `Sequence`'s values without properly respecting the mask.

A `Sequence` is said to be **masked** if for all batch positions b and time positions t, `mask[b, t]` is `False` implies that `values[b, t, ...]` equals zero. The helper method `mask_invalid()` returns a new `Sequence` with this property. Whether a sequence is known to be masked is represented by a marker type `MaskedSequence` which is a subclass of `Sequence` with `mask_invalid()` replaced with a no-op. When critical for performance,

[2] `https://numpy.org/doc/stable/reference/arrays.interface.html`

one can declare a `MaskedSequence` which promises that invalid positions (`mask=False`) are already zeroed out (Figure 2).

`Sequences` have convenience methods to streamline creation, manipulation, and information gathering (e.g., `from_values()`, `from_lengths()`, `lengths()`, `pad_time()`), as well as to support usual shorthands for slicing (`seq[..., a:b]`).

Code

```
x = sl.Sequence(
    jnp.ones((2, 3)),
    jnp.asarray([[True, True, False],
                 [True, False, False]])
)
# Helper that zeroes invalid positions:
x = x.mask_invalid()  # Returns sl.MaskedSequence.
x = x.mask_invalid()  # No-op.
```

Value of `x`:

```
MaskedSequence(
    values=Array([[1., 1., 0.],
                  [1., 0., 0.]],
                 dtype=float32),
    mask=Array([[ True,  True, False],
                [ True, False, False]],
               dtype=bool)
)
```

Figure 2: Example demonstrating the `Sequence` and `MaskedSequence` primitives.

2.2 *Streamable:* The `SequenceLayer` type

`SequenceLayer` is a Python class which defines the basic API and functionality required to achieve the goals of the library.

The fundamental methods of the API are:

- `layer: Sequence -> Sequence`: Process a sequence `x` layer-wise and return a new sequence `y`.
- `get_initial_state: State`: Returns a pytree of state arrays for step-wise execution of the layer.
- `step: (Sequence, State) -> (Sequence, State)`: Processes one block of inputs `x` and produces a block of outputs `y`.

A `SequenceLayer` that is steppable must produce identical outputs for an input sequence regardless of whether the `layer` or `step` API is used (Figure 3).

2.2.1 Output ratio and block size

A `SequenceLayer` can downsample or upsample its input. However, the ratio of output timesteps to input timesteps must be a constant, due to the requirement in compiled JAX or TensorFlow programs that all functions must have fixed shapes and types. This requirement implies that the output shape cannot vary across invocations of any compiled `SequenceLayer` function.

The **output ratio** of a `SequenceLayer` is a constant ratio (represented as a `fractions.Fraction`) between the number of output timesteps and input timesteps to a layer. For example:

Layer	Config	Output Ratio
`Dense`	—	`Fraction(1, 1)`
`Conv1D`	`stride = 1`	`Fraction(1, 1)`
`Conv1D`	`stride = 2`	`Fraction(1, 2)`
`Conv1DTranspose`	`stride = 1`	`Fraction(1, 1)`
`Conv1DTranspose`	`stride = 2`	`Fraction(2, )`
`Downsample1D`	`rate = 2`	`Fraction(1, 2)`
`Upsample1D`	`rate = 2`	`Fraction(2, 1)`
`DotProductSelfAttention`	—	`Fraction(1, 1)`

Code

```python
# Define a test model and input.
model = sl.Conv1D.Config(filters=5, kernel_size=3, padding='causal').make()
x = sl.Sequence.from_values(
  jax.random.uniform(key, (b, t, c))
)

# Process x layer-wise.
y = model.layer(x, training=True)

# Process x step-wise in blocks of size 2.
state = model.get_initial_state(b, x.channel_spec, training=True)
y0, state = model.step(x[:, 0:2], state, training=True)
y1, state = model.step(x[:, 2:4], state, training=True)
y2, state = model.step(x[:, 4:6], state, training=True)
...

y_step = sl.Sequence.concatenate_sequences([y0, y1, y2, ...])

# Layer-wise and step-wise outputs must be equivalent.
np.testing.assert_array_equal(y.values, y_step.values)
np.testing.assert_array_equal(y.mask, y_step.mask)
```

Figure 3: Example demonstrating layer-wise and step-wise execution of a `SequenceLayer`.

Since the input and output shapes of compiled JAX and TensorFlow programs must be constant, a `SequenceLayer step` must always produce at least one output timestep for a group of input timesteps. Since the number of input and output timesteps must be integral and each layer has a constant output ratio, each layer therefore has a **block size** which the total number of input timesteps to `step` must be divisible by in order to produce an integral number of outputs. This is necessarily at least $^1/_{\texttt{output\_ratio}}$, but may be larger (for example, if a layer internally downsamples by 2x and upsamples by 2x, its output ratio would be 1 but its block size would be 2).

2.2.2 State

As shown in Figure 3, `State` is a key part of the step-wise execution of the layer. An initial state is returned from the `get_initial_state` method, then the `state` is repeatedly updated by the `step` method to produce the next output and state for an input block and previous `state`.

The `State` type is any pytree of fixed shape / type arrays representing layer state that evolves over repeated inference calls, such as the KV cache for an attention layer, a buffer for a convolution, or a cell/hidden state for an LSTM.

All SequenceLayer step-wise state is represented via explicit arrays passed into and out of layer methods. No state is stored within the layer (for example via Flax variables[3]). This makes SequenceLayers well suited to the JAX pure functional programming model, despite being implemented in an object oriented manner.

To ensure that `State`s are passed to the correct sublayer of an aggregate SequenceLayer, combinators like `Serial` (Section 2.4.1) wrap and unwrap them into container pytrees in a well-defined and ordered way (e.g., `Serial` takes and returns a `State` which is a tuple of its sublayers' `State`s).

Note that from the perspective of the computation graph compiler (e.g., XLA; Sabne, 2020), there is no difference between data passed via these containers versus the main `Sequence` layer inputs and outputs; rather, these semantic conventions are for the benefit of SequenceLayer users and developers. Hence, if the

[3] `https://flax-linen.readthedocs.io/en/latest/guides/flax_fundamentals/state_params.html`

shapes and dtypes of a `State`'s leaves vary across invocations, this may lead to unnecessary recompilation, just as differently shaped `Sequence` inputs would.

2.2.3 Constants

`Constants` is an optional mutable dictionary of pytrees (`MutableMapping[str, pytree]`) representing auxiliary inputs to the `layer`, `step`, and `get_initial_state` methods. These could be conditioning vectors / sequences which also become part of the computation graph (e.g., during cross-attention), or custom user-defined effects on control flow (e.g., `'causal': False`). The Constants dictionary is mutable and its contents are read and written in the same order that `Sequence` inputs/outputs are propagated within a `SequenceLayer` (e.g., first to last over the layers of a `Serial`), enabling flexible behavior outside guarantees required on the primary sequence data.

For simplicity, constants are available to all layers via the flat namespace of the string keys to the dictionary. Combinators like `Serial` and `Parallel` (Section 2.4) broadcast constants to all sub-layers. This choice comes with the possibility of namespace clashes, but simplifies the API for the common case of providing cross attention sources and conditioning arrays to specific layers. We considered a more complex routing scheme, but decided that it was likely that users could make the keys unique using assumptions about their specific model, which is always easier than building a general-purpose solution that will be infrequently used.

With `Constants`, the API in Section 2.2 becomes:

- **`layer: (Sequence, Constants) -> Sequence`**: Process a sequence `x` layer-wise and return a new sequence `y`.
- **`get_initial_state: Constants -> State`**: Returns a pytree of state arrays for step-wise execution of the layer.
- **`step: (Sequence, State, Constants) -> (Sequence, State)`**: Processes a block of inputs `x` and produces a block of outputs `y`.

2.2.4 Latency and Look-ahead

A `SequenceLayer` can introduce lookahead or delay in its output. A `Delay` layer delays its input by N timesteps, while a `Lookahead` layer drops N inputs to jump forward in the input. Layers like `DotProductSelfAttention` and `Conv1D` support lookback and lookahead as well without shifting the sequence time alignment.

Whatever the behavior of the `layer` method, if the layer supports stepping, the `step` method must produce identical results to be considered a valid `SequenceLayer`. Since input arrives in a stream of multiples of `block_size` timesteps, outputs from `step` may have to wait until enough inputs have arrived to perform the same function as `layer`.

To implement this waiting, a layer may output invalid (`mask = False`) timesteps, waiting for more inputs before it produces a valid timestep. The number of output timesteps the caller must discard before expecting the first valid timestep from a layer is the layer's **output latency**. This is programmatically available via the `output_latency` property of the layer.

As a consequence of the output latency (which may entail internal buffering within the layer `State`), to produce all of the valid outputs from the layer it may be necessary to feed invalid inputs to the layer to "flush" it. The number of inputs required to flush the layer is the layer's **input latency**. This is programmatically available via the `input_latency` property of the layer.

The input and output latency are often the same, but may differ. For example, when a layer upsamples or downsamples its input with lookahead, the input and output latency will be different since the time dimension has different scales in the input sequence and output sequence.

With these definitions in place, we can now correct an oversimplification in Figure 3. Figure 4 demonstrates the full logic required to achieve layer/step equivalence when the layers in use have lookahead or delay.

Code

```python
# A convolution layer with 4 steps of lookahead.
model = sl.Conv1D.Config(filters=3, kernel_size=5, padding='reverse_causal').make()
x = sl.Sequence.from_values(
  jax.random.uniform(key, (b, t, c))
)

# Process x layer-wise.
y = model.layer(x, training=True)

assert model.input_latency == 4
assert model.output_latency == 4

# "Flush" the layer with input_latency invalid timesteps.
x_padded = x.pad_time(0, model.input_latency, valid=False)

y_step, _, _ = sl_utils.step_by_step_dynamic(model, x_padded, training=False)

# Ignore the first output_latency timesteps since they are invalid (mask = False, since the
# first valid output cannot be produced until 4 lookahead timesteps have arrived).
y_step = y_step[:, model.output_latency:]

# Layer-wise and step-wise outputs are equivalent after accounting for latency.
np.testing.assert_array_equal(y.values, y_step.values)
np.testing.assert_array_equal(y.mask, y_step.mask)
```

Figure 4: Demonstration of layer / step equivalence when the `SequenceLayer` has lookahead.

2.2.5 Emits

Since the `layer` and `step` APIs return a single `Sequence` output, there is no easy way for layers to produce auxiliary debugging output.

To support this use case, we introduce additional APIs that return `Emits`. An `Emit` is a layer-defined pytree of arrays or sequences.

The calculation of `Emits` may entail extra work within a layer. While optimizing compilers like XLA (Sabne, 2020) can automatically prune unused computation, we would like to avoid the need for compiling this code in the first place, or the execution of the code in eager mode when the auxiliary outputs will not be used.

To this end, we provide additional optional APIs for computing `layer` and `step` with auxiliary outputs:

- **`layer_with_emits: (Sequence, Constants) -> (Sequence, Emits)`**: Process a sequence `x` layer-wise and return a new sequence `y` and auxiliary emits.
- **`step_with_emits: (Sequence, State, Constants) -> (Sequence, State, Emits)`**: Processes a block of inputs `x` and produces a block of outputs `y` and auxiliary emits.

Additionally, we provide a convenience `Emitting` sub-class of `SequenceLayer` that implements `layer` and `step` in terms of `layer_with_emits` and `step_with_emits`, as well as an `Emit` `SequenceLayer` that simply emits the input to the layer as an emit (for easily "tapping" into the intermediate sequences traversing a stack of layers).

2.2.6 Receptive Field

Having access to accurately computed layer and architecture receptive fields facilitates network design when precise control over model causality is desired. To simplify the calculation of receptive fields for any architecture, we developed the `receptive_field` property. This utility determines the effective range of input time steps that affect a single output time step. When dealing with receptive fields, several key challenges arise:

- **Layers with `output_ratio != 1`**: We define the receptive field as the `(start, end)` tuple that describes the input step range `[t_i + start, t_i + end]` that affects the output time step `t_o`, where `t_i = t_o // output_ratio`.
- **Layers with alternating receptive fields every n steps**: To handle this complexity, we keep track of a step-specific receptive field map via the `receptive_field_per_step` helper property. The overall `receptive_field` is the union of the step-specific receptive fields and describes the maximal temporal window of influence for the layer. `receptive_field_per_step` is used by combinator layers such as `Serial` to compute the overall `receptive_field_per_step` for compositions of layers.
- **Layers with infinite or no receptive field:** The implementation is robust to special cases, representing infinite receptive fields (e.g., `LSTM` has `(-inf, 0)` receptive field) and layers with holes in their receptive field (e.g., `Conv1DTranspose` with `stride > kernel_size` has a `None` receptive field for some time steps).

See Figure 5 for examples of the `receptive_field` property in action.

2.3 *Correct:* The `SequenceLayer` contract

In this section, we introduce the **SequenceLayer contract**, which is the set of requirements a `SequenceLayer` must implement to be considered correct.

- **Layer-wise and Step-wise Equivalence**: If step-wise operation is supported, the `layer` and `step` methods must produce identical results when fed identical data and starting state (slicing the data into blocks of any multiple of `block_size` timesteps. See Figure 3 for a code example. Stateful stochastic layers (e.g., `Dropout`) should obey this property when the starting RNG state is equivalent.
- **Padding and Batching Invariance**: The `layer` and `step` methods must produce identical results when fed identical data **with differing amounts of end padding**, or when **the position of examples in a batch is shuffled**. For the common use-case of batching contiguous sequences of mixed lengths together, the lengths of other sequences in the batch or the position in the batch should have no bearing on the calculation performed by the layer.

  Corollary: Padding values must not affect the calculation of non-padding values.

  Note: Padding invariance is currently only required for end padding. Start padding or interior padding (for non-contiguous sequences) does affect the behavior of calculations. This may change in the future.
- **Masked inputs and outputs**: For an input `Sequence` provided to a `SequenceLayer` with `values` (`[b, t, ...]`) and `mask` (`[b, t]`), layers **must not assume `values` is masked**. If the computation performed by the layer requires masked inputs (e.g., it mixes information across timesteps), then the layer must mask the input sequence before use. The layer may return either a `Sequence` or a `MaskedSequence`.

Each `SequenceLayer` included with the library has unit tests that it obeys this contract. You can test that your own layers obey this contract using the `verify_contract` test utility provided with the library. `verify_contract` performs the following compliance tests:

- `layer` and `step` (taking steps of `block_size` timesteps) produce identical outputs (up to floating point tolerance) on the same input sequence.

Code

```python
# Convolution layer with causal padding.
model = sl.Conv1D.Config(filters=3, kernel_size=5, padding='causal').make()
assert model.receptive_field == (-4, 0)

# Convolution layer with reverse_causal padding.
model = sl.Conv1D.Config(filters=3, kernel_size=5, padding='reverse_causal').make()
assert model.receptive_field == (0, 4)

# Convolution layer with same padding.
model = sl.Conv1D.Config(filters=3, kernel_size=5, padding='same').make()
assert model.receptive_field == (-2, 2)

# Stack of convolution layers with same padding.
model = sl.Serial.Config([Conv1D.Config(filters=3, kernel_size=5, padding='same')] * 4).make()
assert model.receptive_field == (-8, 8)

# LSTM.
model = sl.LSTM.Config(units=3).make()
assert model.receptive_field == (-np.inf, 0)

# Transposed convolution with kernel_size < stride.
model = sl.Conv1DTranspose.Config(filters=1, kernel_size=1, strides=2, padding='same').make()
assert model.receptive_field_per_step == {0: (0, 0), 1: None}
assert model.receptive_field == (0, 0)

# Mixed downsampling + upsampling layers.
model = sl.Serial.Config(
    [
        sl.Conv1D.Config(filters=1, kernel_size=5, strides=2, padding='same'),
        sl.Conv1DTranspose.Config(filters=1, kernel_size=6, strides=4, padding='same'),
    ]
).make()
assert model.receptive_field_per_step == {0: (-4, 2), 1: (-2, 2), 2: (-2, 2), 3: (-2, 4)}
assert model.receptive_field == (-4, 3)
```

Figure 5: Demonstration of the `receptive_field` property for various layers.

- `layer` and `step` produce identical gradients with respect to both their parameters and inputs.
- A step-wise application with $2 \times$ `block_size` produces identical outputs to the layer-wise output.
- The layer's behavior is consistent with its metadata (`get_output_spec`, `input_latency`, `output_latency`, `output_ratio`, `block_size`).
- The `receptive_field` property matches a gradient-based calculation of the receptive field.
- The layer is **batching invariant**, by inserting additional invalid batch items and verifying equivalent output.
- The layer is **padding invariant**, by replacing all invalid timesteps with `NaN`s or large-valued integers and verifying the outputs for valid timesteps are unchanged.

These tests are typically invoked with randomly generated input sequences and layer parameters (taking care to avoid zero-initialization for bias-like parameters).

Additionally, for ease of streaming deployment nearly all layers in the library have unit tests that their `step` function can be exported as a TensorFlow saved model and converted to LiteRT for mobile deployment.

2.4 *Composable:* Declarative API and Combinators

Since every `SequenceLayer` has a uniform API for layer-wise and step-wise processing, it is easy to build **combinators** or `SequenceLayers` that are compositions of other `SequenceLayers`.

2.4.1 The `Serial` Combinator

The simplest combinator is the `Serial` combinator (Figure 6), which simply executes a list of `SequenceLayers` serially.

Code

```python
# Define a serial of 2 causal convolutions.
model = sl.Serial.Config([
  sl.Conv1D.Config(filters=5, kernel_size=3, stride=2, padding='causal'),
  sl.Conv1D.Config(filters=8, kernel_size=5, stride=3, padding='causal'),
]).make()

x = sl.Sequence.from_values(
  jax.random.uniform(key, (b, t, c))
)

# Applies Conv-Conv serially layer-wise and step-wise.
y = model.layer(x, training=True)
y_step, _, _  = sl_utils.step_by_step_dynamic(l, x, training=True)

# Layer-wise and step-wise outputs must be equivalent.
np.testing.assert_array_equal(y.values, y_step.values)
np.testing.assert_array_equal(y.mask, y_step.mask)

# A stride 2 and stride 3 convolution decimates the sequence by 6x.
assert model.output_ratio == fractions.Fraction(1, 6)
assert model.block_size == 6
assert y.shape == (b, t // 6, 8)
```

Figure 6: The `Serial` combinator.

2.4.2 The `Parallel` Combinator

The `Parallel` combinator enables processing an input sequence in parallel by two or more `SequenceLayers`, combining the result at the end according to a fixed number of broadcasted combination strategies (for example, stacking channels, concatenating on the final channel axis, adding channels, averaging across channels).

2.4.3 The `Residual` Combinator

The `Residual` combinator simply transforms a `SequenceLayer` implementing a function $F(x)$ into a residual function $F(x) + x$. Due to the critical importance of residual functions to deep learning (He et al., 2015), for readability this deserves a dedicated combinator even though it can be implemented with `Parallel`.

2.4.4 The `Repeat` Combinator

The `Repeat` combinator (Figure 7) repeats a specified `SequenceLayer` a certain number of times using a control flow primitive such as `jax.lax.scan`. A different set of layer parameters is used for each iteration of the loop.

Using a loop to process the input implies the shape and type of the input is unchanged throughout all iterations of the loop and the calculation performed across all iterations only differs in the inputs and parameters. This is useful for reducing compilation time when compiling large models, since the optimizing compiler only has to compile the function once. This comes with the downsides of not being able to optimize across iterations of the loop. The `unroll_layer` and `unroll_step` options can be used to unroll either the `layer` or `step` operations respectively to enable cross-iteration optimization in either program separately. A `remat` option wraps each iteration of the loop in a gradient checkpoint, so that peak memory usage during backpropagation is reduced.

Code

```
# Repeat TransformerBlock num_blocks times.
model = sl.Repeat.Config(
  TransformerBlock(d_model, ...),
  num_repeats=num_blocks,
  # Checkpoint gradients to reduce memory usage in backpropagation.
  remat=True,
).make()
```

Figure 7: The `Repeat` combinator.

2.4.5 The `Bidirectional` Combinator

The `Bidirectional` combinator processes its input in the forward direction with a `forward` layer, and in the backward direction with a `backward` layer, and then combines the resulting forward and backward sequences. Since this entails reversing the input sequence, it is not steppable. This is effectively a generalization of bidirectional RNNs (Bahdanau et al., 2016) to any forward and backward network.

2.4.6 The `Blockwise` Combinator

The `Blockwise` combinator (Figure 8) is a useful tool for adjusting the block size of any `SequenceLayer` and automatically re-implementing the `layer` method in terms of steps of a chosen `block_size`. This enables adjusting the native streaming block size of a layer without changing any execution code (as long as the execution code uses the `block_size` property of the layer). Additionally, it enables limiting the peak memory usage of the `layer` method by splitting the input sequence into blocks of size `block_size` for processing using the `step` method within `layer`.

3 Implementation

In our initial release, we demonstrate the SequenceLayers design in JAX (using Flax `nn.Module`s; Heek et al., 2024) and TensorFlow 2. The source code is available on GitHub[4], and published on the Python Package Index (PyPI)[5].

See Appendix A for a complete list of layers provided in the JAX implementation at the time of this writing. The TensorFlow version has layers that have not yet been ported to JAX (e.g., `StyleToken`, Wang et al., 2018) and some that have not been backported to TensorFlow (e.g., streaming cross attention layers). This reflects the chronology of our development. SequenceLayers was initially developed as a TensorFlow library, and was ported to JAX as internal research at Google shifted towards JAX.

[4] `https://github.com/google/sequence-layers`
[5] `https://pypi.org/project/sequence-layers/`

Code

```
# Process the input sequence 1024 steps at a time.
model = sl.Blockwise.Config(
  TransformerConfig(d_model, ...),
  block_size=1024,
).make()

assert model.block_size == 1024

# One million timesteps of input, too large to process given our memory constraints.
x = sl.Sequence.from_values(
  jax.random.normal(key, (b, 1000000, d_model))
)

# Layer-wise application of 1 million timesteps streams 1024 steps at a time,
# reducing peak memory usage.
y = model.layer(x, training=True)

# Step-wise application of 1 million timesteps streams 1024 steps at a time,
# increasing throughput versus the default of 1 timestep.
y_step, _, _  = sl_utils.step_by_step_dynamic(l, x, training=True)
```

Figure 8: The `Blockwise` combinator.

4 Discussion

4.1 Eliminating the research to production tax

Scalable machine learning serving systems benefit greatly from abstraction and modularity. TensorFlow Serving (Olston et al., 2017) is able to serve *any* TensorFlow saved model due to the dual abstractions of the **computation graph** and **signatures** that TensorFlow provides. Within Google, no additional infrastructure or model changes are needed to serve a TensorFlow saved model, since it is a well-supported abstraction.

However, TensorFlow signatures are stateless; No state is preserved across requests. The only way to deploy a streaming model on TensorFlow Serving is to plumb the state in and out of the model on every step, which is inefficient and unsuitable for large-scale deployment of models with gigabytes of state, such as Transformer KV caches in billion parameter models or low-latency / realtime scenarios where every microsecond counts.

At Google, serving streaming models at scale has typically necessitated completely custom serving systems designed for the one-off serving of a specific class of model (for example, streaming speech recognition). We refer to the creation of custom serving infrastructure or porting of models from the training implementation into a streamable implementation for serving collectively as **the research to production tax**.

SequenceLayers provides the API abstraction needed to eliminate the research to production tax for most classes of streaming models. At Google, SequenceLayers support has been integrated into standard serving infrastructure, enabling any model built with SequenceLayers to be deployed in production with no model rewriting or custom serving infrastructure.

4.2 Dynamic stream processing pipelines

Many stream processing systems such as MediaPipe (Lugaresi et al., 2019) enable dynamically clocked computation graphs where each node in the computation graph can consume and produce output at different times and rates. For example, nodes in a graph can buffer their inputs while producing no outputs, or block until input from multiple sinks connected to the node are ready before producing an output.

SequenceLayers is not capable of these types of dynamic computations due to the constraints of compiling programs for JAX / TensorFlow to XLA for execution on TPUs and GPUs. These constraints limit SequenceLayers' applicability to these types of problems; however, we suggest that SequenceLayers provides a complementary feature set to dynamic computation graphs, as individual nodes in a dynamic computation graph can be implemented as a SequenceLayer program running on an accelerator.

4.3 Composition and modularity in large research codebases

At Google, SequenceLayers has proven a valuable tool for accelerating research velocity, enabling teams to quickly iterate on the architecture of their models without having to radically change their training and model code as architecture details change.

SequenceLayers has been used to abstract architectural details in:

- Classifiers
- Contrastive / distance metric learning models.
- Regression models.
- Probabilistic models (autoregressive models, normalizing flows, diffusion, VAEs, GANs).

across a wide variety of tasks:

- Audio / speech classification.
- Image classification.
- Contextualized word embedding.
- Text-to-speech synthesis.
- Speech and phoneme recognition.
- Speech translation.
- Speech vocoding.
- Audio tokenization and synthesis.
- Real-time music synthesis.
- Video understanding.
- Language modeling.

The composition capabilities of SequenceLayers has enabled the creation of shared repositories of SequenceLayer definitions for varied purposes. These serve as a building block library that have enabled teams building the above models to easily share code and pre-trained models. Since layer authors adhere to the same API and test their layers for compliance with `verify_contract`, every newly authored `SequenceLayer` is easily reusable and composable.

At time of writing, published and open-sourced work built with SequenceLayers include:

- Gemma 3n's streaming audio encoder[6]
- DolphinGemma[7]
- *Robust and Unbounded Length Generalization in Autoregressive Transformer-Based Text to Speech* (Very Attentive Tacotron; Battenberg et al., 2025)
- *Source Separation by Flow Matching* (Scheibler et al., 2025)
- *Learning the Joint Distribution of Two Sequences using Little or No Paired Data* (Mariooryad et al., 2022)

[6] `https://deepmind.google/models/gemma/gemma-3n`
[7] `https://deepmind.google/models/gemma/dolphingemma`

- *Speaker Generation* (Stanton et al., 2021)
- *Wave-Tacotron: Spectrogram-free End-to-End Text-to-Speech Synthesis* (Weiss et al., 2021)
- Re-implementations of Tacotron (Wang et al., 2017)

4.4 The advantages and disadvantages of modularity

Modularity and abstraction are the cornerstone of not just computer science and engineering, but all forms of engineering. It is impossible to build maintainable large systems or structures without some degree of modularity. Additionally, in large organizations modularity and division of responsibility enables teams crossing timezones to work together effectively to build large systems.

The key to success is to choose the right places to put the abstraction or module boundaries. In some situations, it will be necessary to break an abstraction boundary in the name of efficiency. This may be a sign that a suboptimal boundary was chosen, or simply a sign that the system requirements have changed.

The overall SequenceLayers design is helpful when defining the abstraction boundaries of a system. It can be used to declare a priori that the only thing that matters about a specific sequence-processing block to the broader system it is used within is its underlying sequence-processing properties:

- The shape and type of the input and output sequence.
- The block sizes of inputs that it can operate over.
- Whether the module decimates or upsamples its input.
- The latency (delay or lookahead) in sequence time (not wallclock time) induced by the module.
- The receptive field of the module; the causal relationship between inputs and outputs of the layer.
- The compute profile (the wallclock processing time for its execution, the size of its state arrays, the size of the model parameters, the number of accelerators it requires).

The exact form of the function and the details of its state can be left as a black box. This is powerful for decoupling architectural details (e.g., Transformer vs. RNN) from the role the architecture plays in a broader system (e.g., encode an input sequence non-causally for use by a causal decoder, parameterize the distribution over next-token probabilities given previous samples, predict the noise velocities at the current noise level as part of a diffusion process).

However, there may come a time when the modularity serves as a hindrance. For example, the trend of sharing KV caches across blocks in a Transformer model (Sun et al., 2024) breaks the modularity of the blocks. While it's possible to build a new combinator for this specific style of state sharing, another approach is to simply redraw the abstraction boundary of the `SequenceLayer` to cover the entire Transformer architecture rather than describing each Transformer block as a composition of SequenceLayers. However, from the perspective of consumers of the Transformer nothing has changed and they can continue to abstract the details of the Transformer via a `SequenceLayer`. In our view, the benefits accrued from modularity greatly outweigh the papercut of such a refactoring.

4.5 Training mode

The core `SequenceLayer` API methods (`layer`, `get_initial_state`, `step`) all have a **required** **`training`** keyword argument, which at first glance is an aesthetic wart. In this section we discuss this design decision.

All deep learning layer libraries need to implement some method of changing behavior between training and test time. `Dropout` and `BatchNormalization` are canonical examples that produce this requirement.

Some frameworks opt to represent the specific behavior under control as a call-time parameter, for example Flax (Heek et al., 2024) provides a `deterministic` constructor and call time parameter to `Dropout`. This has the benefit of being explicit about the behavior being controlled. However, any parent layer calling the dropout layer will need a `training` parameter to know how to set this value. The end result is the same, but the interpretation of the `training` flag is left to parent layers instead of the leaf layers (which is useful when the decision is not one-size-fits-all).

Some frameworks such as Keras (Chollet et al., 2018) have a `training` keyword argument to the call method of certain layers (e.g., `keras.Dropout`), **which defaults to `False`**. However, users are not obligated to recursively pass the `training` parameter throughout the call chain of their model. Instead, the `keras.Layer.__call__` method automatically searches upward in the call stack for any manually specified `training` argument, and **defaults to False otherwise**. While this is a nice way of avoiding threading `training` recursively through all layer calls, it comes with significant implementation complexity (Keras owns the `__call__` method) which contributes to the feeling of Keras being a "framework" instead of a library. Our goal is to make SequenceLayers feel like a simple Python library. We therefore discarded this solution.

Some frameworks use a global variable to indicate the training phase (e.g., "`learning_phase`" in TensorFlow Keras). We also discard this solution since global variables introduce complexity around multi-threaded / multi-process Python programs and are generally considered poor software engineering practice.

We also discard any solution that has the possibility of silently incorrect behavior. For example if the user forgets to specify whether we are in the training phase. Silently incorrect behavior is a critical error that could lead to weeks of wasted researcher time. Accordingly, we discard any solution that entails a default value for the `training` argument.

We therefore decided that a required `training` keyword-only argument that must be passed recursively throughout every `SequenceLayer` API that could produce training-specific logic was the simplest way to solve the problem with zero risk of accidental misconfiguration. The cost of threading the `training` argument is a small price to pay, if ugly.

4.6 Configuration Dataclasses

In this section we discuss layer configuration, and the approach taken in the JAX implementation of representing all layer configuration as nested Python dataclasses inheriting from the `SequenceLayerConfig` type.

In contrast, the TensorFlow implementation passes all configuration as arguments to an `__init__` method, as a normal Python program does. We maintain a parallel protocol buffer (Dean et al., 2008) library which mirrors the arguments to `__init__` for constructing an arbitrary composition of `SequenceLayer`s for use in other software modules.

The JAX implementation represents our evolved thinking on the topic of layer configuration, and is our recommended design pattern for implementations of the SequenceLayer design.

The key purposes that configuration serve is to provide a specification of a `SequenceLayer` to construct, without actually creating any objects that might allocate arrays (potentially creating wasted resources or requiring an accelerator to be present).

Benefits of dataclass configuration:

- **Static vs. dynamic values.** Separating layer configuration from layer state (submodules, trainable parameters) enables a clean separation between hyperparameters (Python values, numpy arrays) and JAX objects (Flax `nn.Module`s and `jax.Array`s). This distinction between static (compile time constant) values and dynamic (tracers / symbolic arrays) values is useful (for example, a statically known value can be used to invoke more efficient implementations of an algorithm). We therefore think isolating static and dynamic values reduces the mental burden of having to verify whether a specific value is always statically known.
- **Modularity and composition at SequenceLayer call sites.** The `SequenceLayerConfig` type is essentially a thunk (`Callable[[], SequenceLayer]`) with the added benefit that it can be arbitrarily nested and composed within other `SequenceLayer` types, for example, multiple `SequenceLayerConfig`s can be chained via `Serial.Config`.

Code

```python
import sequence_layers.jax as sl

class ProjectAndAdd42(sl.Stateless):

  class Config(sl.SequenceLayerConfig):
    num_units: int

    # An optional name for the layer.
    name: str | None

    def make(self) -> 'ProjectAndAdd42':
      return ProjectAndAdd42(self, name=self.name)

  config: Config

  def setup(self) -> None:
    self.input_projection = sl.Dense.Config(self.config.num_units).make()
    self.output_projection = sl.Dense.Config(self.config.num_units).make()

  def layer(
    self,
    x: sl.Sequence,
    *,
    training: bool,
    constants: sl.Constants | None = None
  ) -> sl.Sequence:
    y = self.input_projection.layer(x, training=training)
    y = y.apply_values(lambda v: v + 42)
    return self.output_projection.layer(x, training=training)
```

Figure 9: The configuration dataclass pattern in JAX SequenceLayers.

- **Passing a description instead of an instance.** In the event that layer construction entails potentially heavyweight work, passing a description of a layer to be built can avoid accidentally wasted work or errors arising from instantiating a layer outside of its intended environment.

Disadvantages of dataclass configuration

- **Increased boilerplate.** The nested configuration can increase the feeling of `SequenceLayer` implementations being bloated and heavyweight, while we are aiming for the opposite.
- **Copypasta mistakes.** From years of researchers authoring `SequenceLayers`, we have found that the boilerplate entailed by the nested `Config` dataclass pattern can lead to researchers copy-pasting pre-existing layers then modifying them. A common error that the researcher can make is forgetting to update the `make` method of the config dataclass, which returns an instance of the layer the code was copied from instead of the new layer. This has happened often enough that we consider it a real and unfortunate downside of this approach despite the issue ultimately being user error.

5 Conclusion

In this report we introduced SequenceLayers, a neural network layer API and library for sequence modeling designed from the ground up to enable seamless composition and streaming inference. We outlined its core features and how they arise from our design choices, and describe our existing implementation's layers and available architecture definitions.

This library has proved very useful to Google's research and deployment of streaming neural networks. We look forward to your use!

A JAX Layers

At time of writing, we support a broad range of common layers and combinators.

A.1 Combinator Layers

Name	Description
`Bidirectional`	Processes a sequence with separate forward and backward layers and combines their outputs.
`Blockwise`	Processes another layer in fixed-size blocks.
`CheckpointGradient`	Wraps a layer with a gradient checkpoint to save memory during training.
`Parallel`	Applies multiple layers to the same input in parallel and combines their outputs.
`ParallelChannels`	Applies a single shared layer to different groups of channels in the input sequence.
`Repeat`	Applies a single layer multiple times sequentially in a loop.
`Residual`	Creates a residual connection around a sequence of layers, adding the input (with an optional shortcut layer applied) to the output.
`Serial`	Processes an input sequence through a series of layers, one after the other.
`SerialModules`	Similar to `Serial`, but for pre-constructed layer modules.

Table 1: A summary of the available combinator layers.

A.2 Dense and Linear Layers

Name	Description
`Add`	Adds a constant value or array to the input sequence.
`Affine`	Applies a learnable affine transformation (scale and bias) to the input.
`Dense`	A standard fully-connected dense layer that operates on the final dimension of the channel shape.
`DenseShaped`	A dense layer that transforms the input channel shape to a specified output channel shape.
`EinsumDense`	A dense layer that uses an einsum equation to define the transformation between input and output shapes, for example `...ab,ac->...bc`.
`Embedding`	Computes learned vector embeddings for integer-coded inputs.
`EmbeddingTranspose`	A shared-weight transpose of an embedding layer, used for output projection.
`OneHot`	Computes one-hot vectors for integer-coded inputs.
`MaskedDense`	A causally-masked dense layer where each output timestep is a linear projection of all input timesteps at or before the current timestep.
`Scale`	Scales the input sequence by a constant value or array.
`SequenceDense`	A dense layer where a different projection is applied for each timestep.
`SequenceEmbedding`	An embedding layer where a different embedding table is used for each timestep.

Table 2: A summary of the available dense and linear layers.

A.3 Attention Layers

Name	Description
`DotProductSelfAttention`	A multi-headed dot-product self-attention layer with configurable causal masking.
`LocalDotProductSelfAttention`	Identical to `DotProductSelfAttention` in step-wise mode, but with an efficient layer-wise implementation of sliding window attention.
`DotProductAttention`	A standard cross-attention layer that attends to a source sequence from the `constants` dictionary (Section 2.2.3).
`StreamingDotProductAttention`	A cross-attention layer that assumes each call to the `step` method has a different slice of the source sequence provided in the `constants` dictionary (Section 2.2.3).
`StreamingLocalDotProductAttention`	Identical to `StreamingDotProductAttention` in step-wise mode, but with an efficient layer-wise implementation of sliding window attention.
`GmmAttention`	A cross-attention layer that uses a Gaussian Mixture Model to determine where to attend to the source sequence from the `constants` dictionary (Section 2.2.3). Supports monotonic constraints on the location of each component of the mixture.

Table 3: A summary of the available attention layers.

The following relative position embedding schemes are supported. See Table 14 for position embedding layers which can be used as an alternative to these relative embeddings.

Name	Description
`ShawRelativePositionEmbedding`	Computes query-dependent relative position embeddings as described by Shaw et al. (2018).
`T5RelativePositionEmbedding`	Computes relative position biases in the manner of the T5 Transformer (Raffel et al., 2020).
`TransformerXLRelativePositionEmbedding`	Computes relative position embeddings in the manner of Transformer-XL (Dai et al., 2019).

Table 4: A summary of the available relative position embedding layers.

A.4 Convolution-like Layers

Name	Description
Conv1D	A 1D strided or dilated convolution layer.
Conv1DTranspose	A 1D transpose convolution layer for upsampling.
Conv2D	A 2D strided or dilated convolution layer, with the first dimension treated as time.
Conv2DTranspose	A 2D transpose convolution layer for upsampling, with the first dimension treated as time.
Conv3D	A 3D strided or dilated convolution layer, with the first dimension treated as time.
DepthwiseConv1D	A 1D depthwise convolution layer where each input channel is convolved with its own set of filters.
Downsample1D	Downsamples the sequence along the time dimension by taking every Nth element.
Upsample1D	Upsamples the sequence along the time dimension by repetition.
Upsample2D	Upsamples the sequence along the time and one spatial dimension by repetition.

Table 5: A summary of the available convolution layers.

A.5 DSP Layers

Name	Description
Delay	Delays the input sequence by a specified number of timesteps, padding with invalid timesteps.
FFT	Applies a Fast Fourier Transform (FFT) along a specified axis of the input.
Frame	Creates a sequence of overlapping frames from an input sequence.
IFFT	Applies an Inverse Fast Fourier Transform (IFFT) along a specified axis.
IRFFT	Applies an Inverse Real Fast Fourier Transform (IRFFT) to produce a real-valued output.
InverseSTFT	Computes the inverse Short-time Fourier Transform, reconstructing a signal from its spectrogram.
LinearToMelSpectrogram	Converts a linear-scale spectrogram to the mel scale.
Lookahead	Drops a specified number of initial timesteps from the input sequence.
OverlapAdd	Reconstructs a signal by overlapping and adding framed windows.
RFFT	Applies a Real Fast Fourier Transform (RFFT) for real-valued inputs.
STFT	Computes the Short-time Fourier Transform of an input signal.
Window	Applies a window function (e.g., Hann) to the input sequence along a specified axis.

Table 6: A summary of the available DSP layers.

A.6 Recurrent Layers

Name	Description
LSTM	A standard Long Short-Term Memory (LSTM) layer.
RGLRU	A Real-Gated Linear Recurrent Unit (RG-LRU) layer, as used in the Griffin architecture.

Table 7: A summary of the available recurrent layers.

A.7 Utility Layers

Name	Description
`ApplySharding`	Applies sharding annotations to the sequence's values and mask.
`Argmax`	Computes the argmax along the last dimension of the input sequence.
`CheckpointName`	Wraps the layer with a JAX gradient checkpoint name for debugging.
`Dropout`	Applies dropout to the input sequence during training.
`Emit`	An identity layer that emits its input for debugging purposes.
`GradientClipping`	An identity function that clips the gradient's value during backpropagation.
`Lambda`	Wraps a stateless Python lambda function as a SequenceLayer.
`Logging`	A debugging layer that prints information about its inputs during execution.
`OptimizationBarrier`	Applies a JAX optimization barrier to prevent operator fusion.

Table 8: A summary of the available utility layers.

A.8 Conditioning Layers

Name	Description
`Conditioning`	Applies time-synchronized conditioning to an input sequence using a conditioning sequence in the `constants` dictionary (Section 2.2.3).

Table 9: A summary of the available conditioning layers.

A.9 Shape and Type Manipulation Layers

Name	Description
`Cast`	Casts the input sequence to a specified data type.
`EinopsRearrange`	Rearranges the channel dimensions of the input using an `einops.rearrange` equation.
`ExpandDims`	Adds new dimensions of size 1 to the channel shape.
`Flatten`	Flattens all channel dimensions into a single dimension.
`GlobalEinopsRearrange`	Rearranges both the time and channel dimensions using an `einops.rearrange` equation.
`GlobalReshape`	Reshapes both the time and channel dimensions of the sequence.
`MoveAxis`	Moves channel axes to new positions.
`Reshape`	Reshapes the channel dimensions of the input sequence.
`Slice`	Slices the channel dimensions of the input sequence.
`Squeeze`	Removes channel dimensions of size 1.
`SwapAxes`	Swaps two channel axes of the input.
`Transpose`	Permutes the channel dimensions of the input.

Table 10: A summary of the available shape and type manipulation layers.

A.10 Activation and Pointwise Layers

Name	Description
`Abs`	Takes the element-wise absolute value of the input sequence.
`Elu`	Applies the Exponential Linear Unit (ELU) activation function.
`Exp`	Applies the element-wise exponential function to the input.
`GatedLinearUnit`	A Gated Linear Unit (GLU) that halves the channel dimension.
`GatedTanhUnit`	A Gated Tanh Unit that halves the channel dimension.
`GatedUnit`	A generalized gated unit that combines two halves of the input with activations.
`Gelu`	Applies the Gaussian Error Linear Unit (GELU) activation function.
`Identity`	An identity layer that passes its input through unchanged.
`LeakyRelu`	Applies the Leaky Rectified Linear Unit (Leaky ReLU) activation function.
`Log`	Applies the element-wise natural logarithm to the input.
`MaskInvalid`	Replaces invalid (masked) timesteps in the sequence with zeros.
`Maximum`	Performs an element-wise clip with a specified maximum value.
`Minimum`	Performs an element-wise clip with a specified minimum value.
`Mod`	Computes the element-wise remainder of division by a specified divisor.
`PRelu`	Applies a Parametric ReLU where the negative slope is a learnable parameter.
`Power`	Raises the input sequence to a specified power.
`Relu`	Applies the Rectified Linear Unit (ReLU) activation function.
`Sigmoid`	Applies the sigmoid activation function.
`Softmax`	Applies the softmax function along a specified channel axis.
`Softplus`	Applies the softplus activation function.
`Swish`	Applies the Swish activation function.
`Tanh`	Applies the hyperbolic tangent activation function.

Table 11: A summary of the available activation and pointwise layers.

A.11 Normalization Layers

Name	Description
`BatchNormalization`	Applies batch normalization, normalizing across the batch and time dimensions for each feature.
`GroupNormalization`	Applies group normalization, dividing channels into groups and normalizing within each group.
`LayerNormalization`	Applies layer normalization, normalizing over the feature axes for each item in the batch and time.
`RMSNormalization`	Applies Root Mean Square (RMS) normalization, a simplified version of layer normalization without mean-centering.

Table 12: A summary of the available normalization layers.

A.12 Pooling Layers

Name	Description
`AveragePooling1D`	A 1D pooling layer that reduces temporal resolution by taking the average over a sliding window.
`AveragePooling2D`	A 2D pooling layer that reduces temporal and spatial resolution by taking the average over a sliding window.
`MaxPooling1D`	A 1D pooling layer that reduces temporal resolution by taking the maximum over a sliding window.
`MaxPooling2D`	A 2D pooling layer that reduces temporal and spatial resolution by taking the maximum over a sliding window.
`MinPooling1D`	A 1D pooling layer that reduces temporal resolution by taking the minimum over a sliding window.
`MinPooling2D`	A 2D pooling layer that reduces temporal and spatial resolution by taking the minimum over a sliding window.

Table 13: A summary of the available pooling layers.

A.13 Position Embedding Layers

Name	Description
`AddTimingSignal`	Adds sinusoidal timing signals of varying frequencies to the input channels.
`ApplyRotaryPositionalEncoding`	Applies Rotary Positional Encodings (RoPE) to the sequence to provide relative position information.

Table 14: A summary of the available position embedding layers.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, pp. 265–283. USENIX Association, 2016.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016. URL `https://arxiv.org/abs/1409.0473`.

Eric Battenberg, R. J. Skerry-Ryan, Daisy Stanton, Soroosh Mariooryad, Matt Shannon, Julian Salazar, and David Kao. Robust and unbounded length generalization in autoregressive transformer-based text-to-speech. In *NAACL (Long Papers)*, pp. 11789–11806. Association for Computational Linguistics, 2025.

François Chollet et al. Keras: The python deep learning library. *Astrophysics source code library*, pp. ascl–1806, 2018.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc Viet Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *ACL (1)*, pp. 2978–2988. Association for Computational Linguistics, 2019.

Jeff Dean, Sanjay Ghemawat, et al. Protocol buffers, 2008.

Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 4(9), 2018.

Awni Hannun, Jagrit Digani, Angelos Katharopoulos, and Ronan Collobert. MLX: Efficient and flexible machine learning on apple silicon, 2023. URL `https://github.com/ml-explore`.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. URL `https://arxiv.org/abs/1512.03385`.

Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. Flax: A neural network library and ecosystem for JAX, 2024. URL `http://github.com/google/flax`.

Camillo Lugaresi, Jiuqiang Tang, Hadon Nash, Chris McClanahan, Esha Uboweja, Michael Hays, Fan Zhang, Chuo-Ling Chang, Ming Guang Yong, Juhyun Lee, Wan-Teh Chang, Wei Hua, Manfred Georg, and Matthias Grundmann. Mediapipe: A framework for building perception pipelines, 2019. URL `https://arxiv.org/abs/1906.08172`.

Soroosh Mariooryad, Matt Shannon, Siyuan Ma, Tom Bagby, David Kao, Daisy Stanton, Eric Battenberg, and RJ Skerry-Ryan. Learning the joint distribution of two sequences using little or no paired data, 2022. URL `https://arxiv.org/abs/2212.03232`.

Afroz Mohiuddin et al. Trax — deep learning with clear code and speed, 2019. URL `https://github.com/google/trax`.

Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. Tensorflow-serving: Flexible, high-performance ml serving. In *Workshop on ML Systems at NIPS 2017*, 2017.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, pp. 8024–8035, 2019.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020.

Adam Roberts, Hyung Won Chung, Gaurav Mishra, Anselm Levskaya, James Bradbury, Daniel Andor, Sharan Narang, Brian Lester, Colin Gaffney, Afroz Mohiuddin, Curtis Hawthorne, Aitor Lewkowycz, Alex Salcianu, Marc van Zee, Jacob Austin, Sebastian Goodman, Livio Baldini Soares, Haitang Hu, Sasha Tsvyashchenko, Aakanksha Chowdhery, Jasmijn Bastings, Jannis Bulian, Xavier Garcia, Jianmo Ni, Andrew Chen, Kathleen Kenealy, Kehang Han, Michelle Casbon, Jonathan H. Clark, Stephan Lee, Dan Garrette, James Lee-Thorp, Colin Raffel, Noam Shazeer, Marvin Ritter, Maarten Bosma, Alexandre Passos, Jeremy Maitin-Shepard, Noah Fiedel, Mark Omernick, Brennan Saeta, Ryan Sepassi, Alexander Spiridonov, Joshua Newlan, and Andrea Gesmundo. Scaling up models and data with t5x and seqio. *J. Mach. Learn. Res.*, 24:377:1–377:8, 2023.

Amit Sabne. Xla : Compiling machine learning for peak performance, 2020.

Robin Scheibler, John R. Hershey, Arnaud Doucet, and Henry Li. Source separation by flow matching, 2025. URL `https://arxiv.org/abs/2505.16119`.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *NAACL-HLT (2)*, pp. 464–468. Association for Computational Linguistics, 2018.

Jonathan Shen, Patrick Nguyen, Yonghui Wu, Zhifeng Chen, Mia X. Chen, Ye Jia, Anjuli Kannan, Tara Sainath, Yuan Cao, Chung-Cheng Chiu, Yanzhang He, Jan Chorowski, Smit Hinsu, Stella Laurenzo, James Qin, Orhan Firat, Wolfgang Macherey, Suyog Gupta, Ankur Bapna, Shuyuan Zhang, Ruoming Pang, Ron J. Weiss, Rohit Prabhavalkar, Qiao Liang, Benoit Jacob, Bowen Liang, HyoukJoong Lee, Ciprian Chelba, Sébastien Jean, Bo Li, Melvin Johnson, Rohan Anil, Rajat Tibrewal, Xiaobing Liu, Akiko Eriguchi, Navdeep Jaitly, Naveen Ari, Colin Cherry, Parisa Haghani, Otavio Good, Youlong Cheng, Raziel Alvarez, Isaac Caswell, Wei-Ning Hsu, Zongheng Yang, Kuan-Chieh Wang, Ekaterina Gonina, Katrin Tomanek, Ben Vanik, Zelin Wu, Llion Jones, Mike Schuster, Yanping Huang, Dehao Chen, Kazuki Irie, George Foster, John Richardson, Klaus Macherey, Antoine Bruguier, Heiga Zen, Colin Raffel, Shankar Kumar, Kanishka Rao, David Rybach, Matthew Murray, Vijayaditya Peddinti, Maxim Krikun, Michiel A. U. Bacchiani, Thomas B. Jablin, Rob Suderman, Ian Williams, Benjamin Lee, Deepti Bhatia, Justin Carlson, Semih Yavuz, Yu Zhang, Ian McGraw, Max Galkin, Qi Ge, Golan Pundak, Chad Whipkey, Todd Wang, Uri Alon, Dmitry Lepikhin, Ye Tian, Sara Sabour, William Chan, Shubham Toshniwal, Baohua Liao, Michael Nirschl, and Pat Rondon. Lingvo: a modular and scalable framework for sequence-to-sequence modeling, 2019. URL `https://arxiv.org/abs/1902.08295`.

Daisy Stanton, Matt Shannon, Soroosh Mariooryad, RJ Skerry-Ryan, Eric Battenberg, Tom Bagby, and David Kao. Speaker generation, 2021. URL `https://arxiv.org/abs/2111.05095`.

Yutao Sun, Li Dong, Yi Zhu, Shaohan Huang, Wenhui Wang, Shuming Ma, Quanlu Zhang, Jianyong Wang, and Furu Wei. You only cache once: Decoder-decoder architectures for language models, 2024. URL `https://arxiv.org/abs/2405.05254`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, pp. 5998–6008, 2017.

Yuxuan Wang, R. J. Skerry-Ryan, Daisy Stanton, Yonghui Wu, Ron J. Weiss, Navdeep Jaitly, Zongheng Yang, Ying Xiao, Zhifeng Chen, Samy Bengio, Quoc V. Le, Yannis Agiomyrgiannakis, Rob Clark, and Rif A. Saurous. Tacotron: Towards end-to-end speech synthesis. In *INTERSPEECH*, pp. 4006–4010. ISCA, 2017.

Yuxuan Wang, Daisy Stanton, Yu Zhang, RJ Skerry-Ryan, Eric Battenberg, Joel Shor, Ying Xiao, Fei Ren, Ye Jia, and Rif A. Saurous. Style tokens: Unsupervised style modeling, control and transfer in end-to-end speech synthesis, 2018. URL `https://arxiv.org/abs/1803.09017`.

Ron J. Weiss, RJ Skerry-Ryan, Eric Battenberg, Soroosh Mariooryad, and Diederik P. Kingma. Wave-tacotron: Spectrogram-free end-to-end text-to-speech synthesis, 2021. URL `https://arxiv.org/abs/2011.03568`.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. Huggingface's transformers: State-of-the-art natural language processing. *CoRR*, abs/1910.03771, 2019.