trax.models

atari_cnn

Simple net for playing Atari games using PPO.

`trax.models.atari_cnn.AtariCnn`(*n_frames=4, hidden_sizes=(32, 32), output_size=128, mode='train'*)

An Atari CNN.

`trax.models.atari_cnn.AtariCnnBody`(*n_frames=4, hidden_sizes=(32, 64, 64), output_size=512, mode='train', kernel_initializer=None, padding='VALID'*)

An Atari CNN.

`trax.models.atari_cnn.FrameStackMLP`(*n_frames=4, hidden_sizes=(64,), output_size=64, mode='train'*)

MLP operating on a fixed number of last frames.

mlp

mlp – functions that assemble "multilayer perceptron" networks.

`trax.models.mlp.PureMLP`(*layer_widths=(128, 64), activation_fn=<function Relu>, out_activation=False, flatten=True, mode='train'*)

A "multilayer perceptron" (MLP) network.

This is a classic fully connected feedforward network, with one or more layers and a (nonlinear) activation function between each layer. For historical reasons, such networks are often called multilayer perceptrons; but they are more accurately described as multilayer networks, where each layer + activation function is a perceptron-like unit (see, e.g., [https://en.wikipedia.org/wiki/Multilayer_perceptron#Terminology]).

Parameters:

- **layer_widths** – Tuple of ints telling the number of layers and the width of each layer. For example, setting *layer_widths=(128, 64, 32)* would yield 3 layers with successive widths of 128, 64, and 32.
- **activation_fn** – Type of activation function between pairs of fully connected layers; must be an activation-type subclass of *Layer*.
- **out_activation** – If True, include a copy of the activation function as the last layer in the network.
- **flatten** – If True, insert a layer at the head of the network to flatten the input tensor into a matrix of shape (batch_size. -1).
- **mode** – Ignored.

Returns: An assembled MLP network with the specified layers. This network can either be initialized and trained as a full model, or can be used as a building block in a larger network.

`trax.models.mlp.MLP`(*d_hidden=512, n_hidden_layers=2, activation_fn=<function Relu>, n_output_classes=10, flatten=True, mode='train'*)

An MLP network, with a final layer for n-way classification.

neural_gpu

Implementation of the improved Neural GPU (NGPU).

`trax.models.neural_gpu.SaturationCost`(*x, limit=0.9*)

`trax.models.neural_gpu.DiagonalGate`()

Split channels in 3 parts. Shifts 1st and 3rd sections to left/right.

`trax.models.neural_gpu.ConvDiagonalGRU`(*units, kernel_size=(3, 3)*)

Build convolutional GRU with diagonal gating as in ImprovedNGPU.

`trax.models.neural_gpu.NeuralGPU`(*d_feature=96, steps=16, vocab_size=2, mode='train'*)

Implementation of Neural GPU: https://arxiv.org/abs/1702.08727.

Parameters:

- **d_feature** – Number of memory channels (dimensionality of feature embedding).
- **steps** – Number of times depthwise recurrence steps.
- **vocab_size** – Vocabulary size.
- **mode** – Whether we are training or evaluating or doing inference.

Returns: A NeuralGPU Stax model.

resnet

ResNet.

`trax.models.resnet.ConvBlock`(*kernel_size, filters, strides, norm, non_linearity, mode='train'*)

ResNet convolutional striding block.

`trax.models.resnet.IdentityBlock`(*kernel_size, filters, norm, non_linearity, mode='train'*)

ResNet identical size block.

trax.models.resnet.Resnet50(*d_hidden=64, n_output_classes=1001, mode='train', norm=<sphinx.ext.autodoc.importer._MockObject object>, non_linearity=<function Relu>*)

ResNet.

Parameters:
- **d_hidden** – Dimensionality of the first hidden layer (multiplied later).
- **n_output_classes** – Number of distinct output classes.
- **mode** – Whether we are training or evaluating or doing inference.
- **norm** – *Layer* used for normalization, Ex: BatchNorm or FilterResponseNorm.
- **non_linearity** – *Layer* used as a non-linearity, Ex: If norm is BatchNorm then this is a Relu, otherwise for FilterResponseNorm this should be ThresholdedLinearUnit.

Returns: The list of layers comprising a ResNet model with the given parameters.

trax.models.resnet.WideResnetBlock(*channels, strides=(1, 1), bn_momentum=0.9, mode='train'*)

WideResnet convolutional block.

trax.models.resnet.WideResnetGroup(*n, channels, strides=(1, 1), bn_momentum=0.9, mode='train'*)

trax.models.resnet.WideResnet(*n_blocks=3, widen_factor=1, n_output_classes=10, bn_momentum=0.9, mode='train'*)

WideResnet from https://arxiv.org/pdf/1605.07146.pdf.

Parameters:
- **n_blocks** – int, number of blocks in a group. total layers = 6n + 4.
- **widen_factor** – int, widening factor of each group. k=1 is vanilla resnet.
- **n_output_classes** – int, number of distinct output classes.
- **bn_momentum** – float, momentum in BatchNorm.
- **mode** – Whether we are training or evaluating or doing inference.

Returns: The list of layers comprising a WideResnet model with the given parameters.

rl

Policy networks.

trax.models.rl.Policy(*policy_distribution, body=None, normalizer=None, head_init_range=None, batch_axes=None, mode='train'*)

Attaches a policy head to a model body.

trax.models.rl.Value(*body=None, normalizer=None, inject_actions=False, inject_actions_n_layers=1, inject_actions_dim=64, batch_axes=None, mode='train', is_discrete=False, vocab_size=2, multiplicative_action_injection=False*)

Attaches a value head to a model body.

trax.models.rl.PolicyAndValue(*policy_distribution, body=None, policy_top=<function Policy>, value_top=<function Value>, normalizer=None, head_init_range=None, mode='train'*)

Attaches policy and value heads to a model body.

trax.models.rl.Quality(*body=None, normalizer=None, batch_axes=None, mode='train', n_actions=2*)

The network takes as input an observation and outputs values of actions.

rnn

RNNs (recursive neural networks).

trax.models.rnn.RNNLM(*vocab_size, d_model=512, n_layers=2, rnn_cell=<sphinx.ext.autodoc.importer._MockObject object>, rnn_cell_d_state_multiplier=2, dropout=0.1, mode='train'*)

Returns an RNN language model.

This model performs autoregressive language modeling:

- input: rank 2 tensor representing a batch of text strings via token IDs plus padding markers; shape is (batch_size, sequence_length). The tensor elements are integers in *range(vocab_size)*, and *0* values mark padding positions.
- output: rank 3 tensor representing a batch of log-probability distributions for each sequence position over possible token IDs; shape is (batch_size, sequence_length, *vocab_size*).

Parameters:
- **vocab_size** – Input vocabulary size – each element of the input tensor should be an integer in *range(vocab_size)*. These integers typically represent token IDs from a vocabulary-based tokenizer.
- **d_model** – Embedding depth throughout the model.
- **n_layers** – Number of RNN layers.
- **rnn_cell** – Type of RNN cell; must be a subclass of *Layer*.
- **rnn_cell_d_state_multiplier** – Multiplier for feature depth of RNN cell state.
- **dropout** – Stochastic rate (probability) for dropping an activation value when applying dropout.
- **mode** – If *'predict'*, use fast inference; if *'train'* apply dropout.

Returns: An RNN language model as a layer that maps from a tensor of tokens to activations over a vocab set.

trax.models.rnn.GRULM(*vocab_size=256, d_model=512, n_layers=1, mode='train'*)

Returns a GRU (gated recurrent unit) language model.

This model performs autoregressive language modeling:

- input: rank 2 tensor representing a batch of text strings via token IDs plus padding markers; shape is (batch_size, sequence_length). The tensor elements are integers in *range(vocab_size)*, and *0* values mark padding positions.
- output: rank 3 tensor representing a batch of log-probability distributions for each sequence position over possible token IDs; shape is (batch_size, sequence_length, *vocab_size*).

Parameters:
- **vocab_size** – Input vocabulary size – each element of the input tensor should be an integer in *range(vocab_size)*. These integers typically represent token IDs from a vocabulary-based tokenizer.
- **d_model** – Embedding depth throughout the model.
- **n_layers** – Number of GRU layers.
- **mode** – If *'predict'*, use fast inference (and omit the right shift).

Returns: A GRU language model as a layer that maps from a tensor of tokens to activations over a vocab set.

`trax.models.rnn.LSTMSeq2SeqAttn`(*input_vocab_size=256, target_vocab_size=256, d_model=512, n_encoder_layers=2, n_decoder_layers=2, n_attention_heads=1, attention_dropout=0.0, mode='train'*)

Returns an LSTM sequence-to-sequence model with attention.

This model is an encoder-decoder that performs tokenized string-to-string ("source"-to-"target") transduction:

- inputs (2):
  - source: rank 2 tensor representing a batch of text strings via token IDs plus padding markers; shape is (batch_size, sequence_length). The tensor elements are integers in *range(input_vocab_size)*, and *0* values mark padding positions.
  - target: rank 2 tensor representing a batch of text strings via token IDs plus padding markers; shape is (batch_size, sequence_length). The tensor elements are integers in *range(output_vocab_size)*, and *0* values mark padding positions.
- output: rank 3 tensor representing a batch of log-probability distributions for each sequence position over possible token IDs; shape is (batch_size, sequence_length, *vocab_size*).

An example use would be to translate (tokenized) sentences from English to German.

The model works as follows:

- Input encoder runs on the input tokens and creates activations that are used as both keys and values in attention.
- Pre-attention decoder runs on the targets and creates activations that are used as queries in attention.
- Attention runs on the queries, keys and values masking out input padding.
- Decoder runs on the result, followed by a cross-entropy loss.

Parameters:
- **input_vocab_size** – Input vocabulary size – each element of the input tensor should be an integer in *range(vocab_size)*. These integers typically represent token IDs from a vocabulary-based tokenizer.
- **target_vocab_size** – Target vocabulary size.
- **d_model** – Final dimension of tensors at most points in the model, including the initial embedding output.
- **n_encoder_layers** – Number of LSTM layers in the encoder.
- **n_decoder_layers** – Number of LSTM layers in the decoder after attention.
- **n_attention_heads** – Number of attention heads.
- **attention_dropout** – Stochastic rate (probability) for dropping an activation value when applying dropout within an attention block.
- **mode** – If *'predict'*, use fast inference. If *'train'*, each attention block will include dropout; else, it will pass all values through unaltered.

Returns: An LSTM sequence-to-sequence model as a layer that maps from a source-target tokenized text pair to activations over a vocab set.

transformer

Transformer models: encoder, decoder, language model, and encoder-decoder.

The "Transformer" name and network architecture were introduced in the paper [Attention Is All You Need](https://arxiv.org/abs/1706.03762).

`trax.models.transformer.TransformerEncoder`(*vocab_size, n_classes=10, d_model=512, d_ff=2048, n_layers=6, n_heads=8, max_len=2048, dropout=0.1, dropout_shared_axes=None, mode='train', ff_activation=<function Relu>*)

Returns a Transformer encoder merged with an N-way categorization head.

This model performs text categorization:

- input: rank 2 tensor representing a batch of text strings via token IDs plus padding markers; shape is (batch_size, sequence_length). The tensor elements are integers in *range(vocab_size)*, and *0* values mark padding positions.
- output: rank 2 tensor representing a batch of log-probability distributions over N categories; shape is (batch_size, *n_classes*).

Parameters:

- **vocab_size** – Input vocabulary size – each element of the input tensor should be an integer in *range(vocab_size)*. These integers typically represent token IDs from a vocabulary-based tokenizer.
- **n_classes** – Final dimension of the output tensors, representing N-way classification.
- **d_model** – Final dimension of tensors at most points in the model, including the initial embedding output.
- **d_ff** – Size of special dense layer in the feed-forward part of each encoder block.
- **n_layers** – Number of encoder blocks. Each block includes attention, dropout, residual, feed-forward (*Dense*), and activation layers.
- **n_heads** – Number of attention heads.
- **max_len** – Maximum symbol length for positional encoding.
- **dropout** – Stochastic rate (probability) for dropping an activation value when applying dropout within an encoder block.
- **dropout_shared_axes** – Tensor axes on which to share a dropout mask. Sharing along batch and sequence axes (*dropout_shared_axes=(0,1)*) is a useful way to save memory and apply consistent masks to activation vectors at different sequence positions.
- **mode** – If *'train'*, each encoder block will include dropout; else, it will pass all values through unaltered.
- **ff_activation** – Type of activation function at the end of each encoder block; must be an activation-type subclass of *Layer*.

Returns: A Transformer model that maps strings (conveyed via token IDs) to probability-like activations over a range of output classes.

`trax.models.transformer.TransformerDecoder`(*vocab_size=None, d_model=512, d_ff=2048, n_layers=6, n_heads=8, max_len=2048, dropout=0.1, dropout_shared_axes=None, mode='train', ff_activation=<function Relu>*)

Returns a Transformer decoder.

This model maps sequential inputs to sequential outputs:

- input if *vocab_size* is specified: rank 2 tensor representing a batch of text strings via token IDs plus padding markers; shape is (batch_size, sequence_length). The tensor elements are integers in *range(vocab_size)*, and *0* values mark padding positions.
- input if *vocab_size* is None: rank 2 tensor representing a batch of activation vectors; shape is (batch_size, sequence_length, *d_model*).
- output: rank 3 tensor with shape (batch_size, sequence_length, *d_model*).

The model uses causal attention and does *not* shift the input to the right. Thus, the output for position *t* is based on inputs up to and including position *t*.

Parameters:

- **vocab_size** – If specified, gives the input vocabulary size – each element of the input tensor should be an integer in *range(vocab_size)*. If None, indicates that the model expects as input floating point vectors, each with *d_model* components.
- **d_model** – Final dimension of tensors at most points in the model, including the initial embedding output.
- **d_ff** – Size of special dense layer in the feed-forward part of each decoder block.
- **n_layers** – Number of decoder blocks. Each block includes attention, dropout, residual, feed-forward (*Dense*), and activation layers.
- **n_heads** – Number of attention heads.
- **max_len** – Maximum symbol length for positional encoding.
- **dropout** – Stochastic rate (probability) for dropping an activation value when applying dropout within a decoder block.
- **dropout_shared_axes** – Tensor axes on which to share a dropout mask. Sharing along batch and sequence axes (*dropout_shared_axes=(0,1)*) is a useful way to save memory and apply consistent masks to activation vectors at different sequence positions.
- **mode** – If *'train'*, each decoder block will include dropout; else, it will pass all values through unaltered.
- **ff_activation** – Type of activation function at the end of each decoder block; must be an activation-type subclass of *Layer*.

Returns:

a Transformer model that maps strings (conveyed via token IDs) to sequences of activation vectors.

If *vocab_size* is None: a Transformer model that maps sequences of activation vectors to sequences of activation vectors.

Return type: If *vocab_size* is defined

`trax.models.transformer.TransformerLM`(*vocab_size, d_model=512, d_ff=2048, n_layers=6, n_heads=8, max_len=2048, dropout=0.1, dropout_shared_axes=None, mode='train', ff_activation=<function Relu>*)

Returns a Transformer language model.

This model performs autoregressive language modeling:

- input: rank 2 tensor representing a batch of text strings via token IDs plus padding markers; shape is (batch_size, sequence_length). The tensor elements are integers in *range(vocab_size)*, and *0* values mark padding positions.
- output: rank 3 tensor representing a batch of log-probability distributions for each sequence position over possible token IDs; shape is (batch_size, sequence_length, *vocab_size*).

This model uses only the decoder part of the overall Transformer.

Parameters:

- **vocab_size** – Input vocabulary size – each element of the input tensor should be an integer in *range(vocab_size)*. These integers typically represent token IDs from a vocabulary-based tokenizer.
- **d_model** – Final dimension of tensors at most points in the model, including the initial embedding output.
- **d_ff** – Size of special dense layer in the feed-forward part of each encoder block.
- **n_layers** – Number of encoder blocks. Each block includes attention, dropout, residual, feed-forward (*Dense*), and activation layers.
- **n_heads** – Number of attention heads.
- **max_len** – Maximum symbol length for positional encoding.
- **dropout** – Stochastic rate (probability) for dropping an activation value when applying dropout within an encoder block.
- **dropout_shared_axes** – Tensor axes on which to share a dropout mask. Sharing along batch and sequence axes (*dropout_shared_axes=(0,1)*) is a useful way to save memory and apply consistent masks to activation vectors at different sequence positions.
- **mode** – If *'predict'*, use fast inference. If *'train'*, each encoder block will include dropout; else, it will pass all values through unaltered.
- **ff_activation** – Type of activation function at the end of each encoder block; must be an activation-type subclass of *Layer*.

Returns: A Transformer language model as a layer that maps from a tensor of tokens to activations over a vocab set.

`trax.models.transformer.Transformer`(*input_vocab_size, output_vocab_size=None, d_model=512, d_ff=2048, n_encoder_layers=6, n_decoder_layers=6, n_heads=8, max_len=2048, dropout=0.1, dropout_shared_axes=None, mode='train', ff_activation=<function Relu>*)

Returns a full Transformer model.

This model is an encoder-decoder that performs tokenized string-to-string ("source"-to-"target") transduction:

- inputs (2):
  - source: rank 2 tensor representing a batch of text strings via token IDs plus padding markers; shape is (batch_size, sequence_length). The tensor elements are integers in *range(input_vocab_size)*, and *0* values mark padding positions.
  - target: rank 2 tensor representing a batch of text strings via token IDs plus padding markers; shape is (batch_size, sequence_length). The tensor elements are integers in *range(output_vocab_size)*, and *0* values mark padding positions.
- output: rank 3 tensor representing a batch of log-probability distributions for each sequence position over possible token IDs; shape is (batch_size, sequence_length, *vocab_size*).

An example use would be to translate (tokenized) sentences from English to German.

Parameters:

- **input_vocab_size** – Input vocabulary size – each element of the input tensor should be an integer in *range(vocab_size)*. These integers typically represent token IDs from a vocabulary-based tokenizer.
- **output_vocab_size** – If specified, gives the vocabulary size for the targets; if None, then input and target integers (token IDs) are assumed to come from the same vocabulary.
- **d_model** – Final dimension of tensors at most points in the model, including the initial embedding output.
- **d_ff** – Size of special dense layer in the feed-forward part of each encoder and decoder block.
- **n_encoder_layers** – Number of encoder blocks.
- **n_decoder_layers** – Number of decoder blocks.
- **n_heads** – Number of attention heads.
- **max_len** – Maximum symbol length for positional encoding.
- **dropout** – Stochastic rate (probability) for dropping an activation value when applying dropout within an encoder/decoder block.
- **dropout_shared_axes** – Tensor axes on which to share a dropout mask. Sharing along batch and sequence axes (*dropout_shared_axes=(0,1)*) is a useful way to save memory and apply consistent masks to activation vectors at different sequence positions.
- **mode** – If *'predict'*, use fast inference. If *'train'*, each encoder/decoder block will include dropout; else, it will pass all values through unaltered.
- **ff_activation** – Type of activation function at the end of each encoder/decoder block; must be an activation-type subclass of *Layer*.

Returns: A Transformer model as a layer that maps from a source-target tokenized text pair to activations over a vocab set.

reformer.reformer

Reformer Models.

`trax.models.reformer.reformer.FeedForward`(*d_model, d_ff, dropout, activation, act_dropout, mode*)

Feed-forward block with layer normalization at start.

`trax.models.reformer.reformer.ChunkedFeedForward`(*d_model, d_ff, dropout, activation, act_dropout, chunk_size, mode*)

Chunked feed-forward block with layer normalization at start.

`trax.models.reformer.reformer.FeedForwardWithOptions`(*d_model, d_ff, dropout, ff_activation, ff_dropout, ff_chunk_size, ff_use_sru, ff_sparsity, mode*)

Feed-Forward block with all the options.

`trax.models.reformer.reformer.DecoderBlock`(*d_model, d_ff, d_attention_key, d_attention_value,* *n_heads, attention_type, dropout, ff_activation, ff_dropout, ff_use_sru, ff_chunk_size, ff_sparsity, mode)*

Reversible transformer decoder layer.

Parameters:

- **d_model** – int: depth of embedding
- **d_ff** – int: depth of feed-forward layer
- **d_attention_key** – int: depth of key vector for each attention head
- **d_attention_value** – int: depth of value vector for each attention head
- **n_heads** – int: number of attention heads
- **attention_type** – subclass of tl.BaseCausalAttention: attention class to use
- **dropout** – float: dropout rate (how much to drop out)
- **ff_activation** – the non-linearity in feed-forward layer
- **ff_dropout** – the dropout rate in feed-forward layer
- **ff_use_sru** – int; if > 0, we use this many SRU layers instead of feed-forward
- **ff_chunk_size** – int; if > 0, chunk feed-forward into this-sized chunks
- **ff_sparsity** – int, if > 0 use sparse feed-forward block with this sparsity
- **mode** – str: 'train' or 'eval'

Returns: the layer.

`trax.models.reformer.reformer.PositionalEncoding`*(mode, dropout=None, max_len=None, axial_pos_shape=None, d_axial_pos_embs=None)*

Returns the positional encoding layer depending on the arguments.

`trax.models.reformer.reformer.ReformerLM`*(vocab_size, d_model=512, d_ff=2048, d_attention_key=64, d_attention_value=64, n_layers=6, n_heads=8, dropout=0.1, max_len=2048, attention_type=<sphinx.ext.autodoc.importer._MockObject object>, axial_pos_shape=(), d_axial_pos_embs=None, ff_activation=<function FastGelu>, ff_use_sru=0, ff_chunk_size=0, ff_sparsity=0, mode='train')*

Reversible transformer language model (only uses a decoder, no encoder).

Parameters:

- **vocab_size** – int: vocab size
- **d_model** – int: depth of *each half* of the two-part features
- **d_ff** – int: depth of feed-forward layer
- **d_attention_key** – int: depth of key vector for each attention head
- **d_attention_value** – int: depth of value vector for each attention head
- **n_layers** – int: number of decoder layers
- **n_heads** – int: number of attention heads
- **dropout** – float: dropout rate (how much to drop out)
- **max_len** – int: maximum symbol length for positional encoding
- **attention_type** – class: attention class to use, such as SelfAttention.
- **axial_pos_shape** – tuple of ints: input shape to use for the axial position encoding. If unset, axial position encoding is disabled.
- **d_axial_pos_embs** – tuple of ints: depth of position embedding for each axis. Tuple length must match axial_pos_shape, and values must sum to d_model.
- **ff_activation** – the non-linearity in feed-forward layer
- **ff_use_sru** – int; if > 0, we use this many SRU layers instead of feed-forward
- **ff_chunk_size** – int; if > 0, chunk feed-forward into this-sized chunks
- **ff_sparsity** – int, if > 0 use sparse feed-forward block with this sparsity
- **mode** – str: 'train', 'eval', or 'predict'

Returns: the layer.

`trax.models.reformer.reformer.ReformerShortenLM`*(vocab_size, shorten_factor=1, d_embedding=256, d_model=512, d_ff=2048, d_attention_key=64, d_attention_value=64, n_layers=6, n_heads=8, dropout=0.1, max_len=2048, attention_type=<sphinx.ext.autodoc.importer._MockObject object>, axial_pos_shape=(), d_axial_pos_embs=None, ff_activation=<function FastGelu>, ff_use_sru=0, ff_chunk_size=0, ff_sparsity=0, mode='train')*

Reversible transformer language model with shortening.

When shorten_factor is F and processing an input of shape [batch, length], we embed the (shifted-right) input and then group each F elements (on length) into a single vector – so that in the end we process a tensor of shape

```
[batch, length // F, d_model]
```

almost until the end – at the end it's un-shortend and a SRU is applied. This reduces the length processed inside the main model body, effectively making the model faster but possibly slightly less accurate.

Parameters:

- **vocab_size** – int: vocab size
- **shorten_factor** – by how much to shorten, see above
- **d_embedding** – the depth of the embedding layer and final logits
- **d_model** – int: depth of *each half* of the two-part features
- **d_ff** – int: depth of feed-forward layer
- **d_attention_key** – int: depth of key vector for each attention head
- **d_attention_value** – int: depth of value vector for each attention head
- **n_layers** – int: number of decoder layers
- **n_heads** – int: number of attention heads
- **dropout** – float: dropout rate (how much to drop out)
- **max_len** – int: maximum symbol length for positional encoding
- **attention_type** – class: attention class to use, such as SelfAttention.
- **axial_pos_shape** – tuple of ints: input shape to use for the axial position encoding. If unset, axial position encoding is disabled.
- **d_axial_pos_embs** – tuple of ints: depth of position embedding for each axis. Tuple length must match axial_pos_shape, values must sum to d_embedding.
- **ff_activation** – the non-linearity in feed-forward layer
- **ff_use_sru** – int; if > 0, we use this many SRU layers instead of feed-forward
- **ff_chunk_size** – int; if > 0, chunk feed-forward into this-sized chunks
- **ff_sparsity** – int, if > 0 use sparse feed-forward block with this sparsity
- **mode** – str: 'train' or 'eval'

Returns: the layer.

`trax.models.reformer.reformer.EncoderBlock`(*d_model, d_ff, n_heads, attention_type, dropout, ff_activation, ff_dropout, ff_use_sru=0, ff_chunk_size=0, ff_sparsity=0, mode='train'*)

Returns a list of layers that implements a Reformer encoder block.

The input to the layer is a pair, (activations, mask), where the mask was created from the original source tokens to prevent attending to the padding part of the input.

Parameters:

- **d_model** – int: depth of embedding
- **d_ff** – int: depth of feed-forward layer
- **n_heads** – int: number of attention heads
- **attention_type** – subclass of tl.BaseCausalAttention: attention class to use
- **dropout** – float: dropout rate (how much to drop out)
- **ff_activation** – the non-linearity in feed-forward layer
- **ff_dropout** – the dropout rate in feed-forward layer
- **ff_use_sru** – int; if > 0, we use this many SRU layers instead of feed-forward
- **ff_chunk_size** – int; if > 0, chunk feed-forward into this-sized chunks
- **ff_sparsity** – int, if > 0 use sparse feed-forward block with this sparsity
- **mode** – str: 'train' or 'eval'

Returns: A list of layers that maps (activations, mask) to (activations, mask).

`trax.models.reformer.reformer.EncoderDecoderBlock`(*d_model, d_ff, n_heads, dropout, ff_activation, ff_dropout, mode*)

Reversible transformer decoder layer.

Parameters:

- **d_model** – int: depth of embedding
- **d_ff** – int: depth of feed-forward layer
- **n_heads** – int: number of attention heads
- **dropout** – float: dropout rate (how much to drop out)
- **ff_activation** – the non-linearity in feed-forward layer
- **ff_dropout** – float: (optional) separate dropout rate for feed-forward layer
- **mode** – str: 'train' or 'eval'

Returns: the layer.

`trax.models.reformer.reformer.Reformer`(*input_vocab_size, output_vocab_size=None, d_model=512, d_ff=2048, n_encoder_layers=6, n_decoder_layers=6, n_heads=8, dropout=0.1, max_len=2048, ff_activation=<function Relu>, ff_dropout=None, mode='train'*)

Reversible transformer encoder-decoder model.

This model expects an input pair: target, source.

At the moment, this model supports dot-product attention only. For the attention types in the Reformer paper, see ReformerLM.

Parameters:

- **input_vocab_size** – int: vocab size of the source.
- **output_vocab_size** – int (optional): vocab size of the target. If None, the source and target are assumed to have the same vocab.
- **d_model** – int: depth of embedding
- **d_ff** – int: depth of feed-forward layer
- **n_encoder_layers** – int: number of encoder layers
- **n_decoder_layers** – int: number of decoder layers
- **n_heads** – int: number of attention heads
- **dropout** – float: dropout rate (how much to drop out)
- **max_len** – int: maximum symbol length for positional encoding
- **ff_activation** – the non-linearity in feed-forward layer
- **ff_dropout** – float: (optional) separate dropout rate at feed-forward nonlinearity. This is called relu_dropout in T2T.
- **mode** – str: 'train' or 'eval'

Returns: A Reformer model as a layer that maps from a target, source pair to activations over a vocab set.

`trax.models.reformer.reformer.Reformer2`(*input_vocab_size, output_vocab_size=None, d_model=512, d_ff=2048, d_attention_key=None, d_attention_value=None, n_encoder_layers=6, n_decoder_layers=6, n_heads=8, dropout=0.1, max_len=2048, encoder_attention_type=<sphinx.ext.autodoc.importer._MockObject object>, encoder_decoder_attention_type=<sphinx.ext.autodoc.importer._MockObject object>, axial_pos_shape='fixed-base', d_axial_pos_embs=None, ff_activation=<function Relu>, ff_use_sru=0, ff_chunk_size=0, ff_dropout=None, ff_sparsity=0, n_layers_forget=0, mode='train'*)

Reversible transformer encoder-decoder model.

This model expects an input pair: source, target.

At the moment, this model supports dot-product attention only. For the attention types in the Reformer paper, see ReformerLM.

Parameters:

- **input_vocab_size** – int: vocab size of the source.
- **output_vocab_size** – int (optional): vocab size of the target. If None, the source and target are assumed to have the same vocab.
- **d_model** – int: depth of embedding
- **d_ff** – int: depth of feed-forward layer
- **d_attention_key** – int: depth of key vector for each attention head
- **d_attention_value** – int: depth of value vector for each attention head
- **n_encoder_layers** – int: number of encoder layers
- **n_decoder_layers** – int: number of decoder layers
- **n_heads** – int: number of attention heads
- **dropout** – float: dropout rate (how much to drop out)
- **max_len** – int: maximum symbol length for positional encoding
- **encoder_attention_type** – class: attention class to use, such as SelfAttention
- **encoder_decoder_attention_type** – class: attention class to use, such as SelfAttention
- **axial_pos_shape** – tuple of ints: input shape to use for the axial position encoding. If unset, axial position encoding is disabled.
- **d_axial_pos_embs** – tuple of ints: depth of position embedding for each axis. Tuple length must match axial_pos_shape, and values must sum to d_model.
- **ff_activation** – the non-linearity in feed-forward layer
- **ff_use_sru** – int; if > 0, we use this many SRU layers instead of feed-forward
- **ff_chunk_size** – int; if > 0, chunk feed-forward into this-sized chunks
- **ff_dropout** – float: (optional) separate dropout rate at feed-forward nonlinearity. This is called relu_dropout in T2T.
- **ff_sparsity** – int, if > 0 use sparse feed-forward block with this sparsity
- **n_layers_forget** – how often to have a forgetting block between layers
- **mode** – str: 'train' or 'eval'

Returns: A Reformer model as a layer that maps from a target, source pair to activations over a vocab set.

research.bert

BERT.

class `trax.models.research.bert.AddBias`(*n_in=1, n_out=1, name=None, sublayers_to_print=None*)

Bases: `trax.layers.base.Layer`

`forward`(*inputs*)

Computes this layer's output as part of a forward pass through the model.

Authors of new layer subclasses should override this method to define the forward computation that their layer performs. Use *self.weights* to access trainable weights of this layer. If you need to use local non-trainable state or randomness, use *self.rng* for the random seed (no need to set it) and use *self.state* for non-trainable state (and set it to the new value).

Parameters: **inputs** – Zero or more input tensors, packaged as described in the *Layer* class docstring.

Returns: Zero or more output tensors, packaged as described in the *Layer* class docstring.

`init_weights_and_state`(*input_signature*)

Initializes weights and state for inputs with the given signature.

Authors of new layer subclasses should override this method if their layer uses trainable weights or non-trainable state. To initialize trainable weights, set *self.weights* and to initialize non-trainable state, set *self.state* to the intended value.

Parameters: **input_signature** – A *ShapeDtype* instance (if this layer takes one input) or a list/tuple of *ShapeDtype* instances; signatures of inputs.

`trax.models.research.bert.BERTClassifierHead`(*n_classes*)

`trax.models.research.bert.BERTRegressionHead`()

`trax.models.research.bert.BERT`(*d_model=768, vocab_size=30522, max_len=512, type_vocab_size=2, n_heads=12, d_ff=3072, n_layers=12, head=None, init_checkpoint=None, mode='eval'*)

BERT (default hparams are for bert-base-uncased).

class `trax.models.research.bert.PretrainedBERT`(**sublayers, init_checkpoint=None*)

Bases: `trax.layers.combinators.Serial`

Wrapper that always initializes weights from a pre-trained checkpoint.

`__init__`(**sublayers, init_checkpoint=None*)

Creates a partially initialized, unconnected layer instance.

Parameters:

- **n_in** – Number of inputs expected by this layer.
- **n_out** – Number of outputs promised by this layer.
- **name** – Class-like name for this layer; for use when printing this layer.
- **sublayers_to_print** – Sublayers to display when printing out this layer; By default (when None) we display all sublayers.

`new_weights`(*input_signature*)

research.skipping_transformer