Layers

Base

LayerType

class paddle.trainer_config_helpers.layers.**LayerType**

Layer type enumerations.

static **is_layer_type**(*type_name*)

If type_name is a layer type.

Parameters:	**type_name** (*basestring*) — layer type name. Because layer type enumerations are strings.
Returns:	True if is a layer_type
Return type:	bool

LayerOutput

class paddle.trainer_config_helpers.layers.**LayerOutput**(*name*, *layer_type*, *parents=None*, *activation=None*, *num_filters=None*, *img_norm_type=None*, *size=None*, *outputs=None*, *reverse=None*)

LayerOutput is output for layer function. It is used internally by several reasons.

- Check layer connection make sense.
    - FC(Softmax) => Cost(MSE Error) is not good for example.
- Tracking layer connection.
- Pass to layer methods as input.

Parameters:

- **name** (*basestring*) — Layer output name.
- **layer_type** (*basestring*) — Current Layer Type. One of LayerType enumeration.
- **activation** (*BaseActivation.*) — Layer Activation.
- **parents** (*list|tuple|collections.Sequence*) — Layer's parents.

Data layer

data_layer

paddle.trainer_config_helpers.layers.**data_layer**(**args*, ***kwargs*)

Define DataLayer For NeuralNetwork.

The example usage is:

```
data = data_layer(name="input",
                  size=1000)
```

Parameters:
- **name** (*basestring*) — Name of this data layer.
- **size** (*int*) — Size of this data layer.
- **layer_attr** (*ExtraLayerAttribute.*) — Extra Layer Attribute.

Returns: LayerOutput object.

Return type: LayerOutput

Fully Connected Layers

fc_layer

paddle.trainer_config_helpers.layers.**fc_layer**(**args*, ***kwargs*)

Helper for declare fully connected layer.

The example usage is:

```
fc = fc_layer(input=layer,
              size=1024,
              act=LinearActivation(),
              bias_attr=False)
```

which is equal to:

```
with mixed_layer(size=1024) as fc:
    fc += full_matrix_projection(input=layer)
```

Parameters:
- **name** (*basestring*) — The Layer Name.
- **input** (*LayerOutput|list|tuple*) — The input layer. Could be a list/tuple of input layer.
- **size** (*int*) — The layer dimension.
- **act** (*BaseActivation*) — Activation Type. Default is tanh.
- **param_attr** (*ParameterAttribute*) — The Parameter Attribute|list.
- **bias_attr** (*ParameterAttribute|None|Any*) — The Bias Attribute. If no bias, then pass False or something not type of ParameterAttribute. None will get a default Bias.
- **layer_attr** (*ExtraLayerAttribute|None*) — Extra Layer config.

Returns: LayerOutput object.

Return type: LayerOutput

selective_fc_layer

paddle.trainer_config_helpers.layers.**selective_fc_layer**(**args*, ***kwargs*)

Selectived fully connected layer. Different from fc_layer, the output of this layer maybe sparse. It requires an additional input to indicate several selected columns for output. If the selected columns is not specified, selective_fc_layer acts exactly like fc_layer.

The simple usage is:

```
sel_fc = selective_fc_layer(input=input, size=128, act=TanhActivation())
```

Parameters:
- **name** (*basestring*) — The Layer Name.
- **input** (*LayerOutput|list|tuple*) — The input layer.
- **select** (*LayerOutput*) — The select layer. The output of select layer should be a sparse binary matrix, and treat as the mask of selective fc.
- **size** (*int*) — The layer dimension.
- **act** (*BaseActivation*) — Activation Type. Default is tanh.
- **param_attr** (*ParameterAttribute*) — The Parameter Attribute.
- **bias_attr** (*ParameterAttribute|None|Any*) — The Bias Attribute. If no bias, then pass False or something not type of ParameterAttribute. None will get a default Bias.
- **layer_attr** (*ExtraLayerAttribute|None*) — Extra Layer config.

Returns: LayerOutput object.

Return type: LayerOutput

Conv Layers

conv_operator

paddle.trainer_config_helpers.layers.**conv_operator**(*img*, *filter*, *filter_size*, *num_filters*, *num_channels=None*, *stride=1*, *padding=0*, *filter_size_y=None*, *stride_y=None*, *padding_y=None*)

Different from img_conv_layer, conv_op is an Operator, which can be used in mixed_layer. And conv_op takes two inputs to perform convolution. The first input is the image and the second is filter kernel. It only support GPU mode.

The example usage is:

```
op = conv_operator(img=input1,
                   filter=input2,
                   filter_size=3,
                   num_filters=64,
                   num_channels=64)
```

Parameters:
- **img** (*LayerOutput*) — input image
- **filter** (*LayerOutput*) — input filter
- **filter_size** (*int*) — The x dimension of a filter kernel.
- **filter_size_y** (*int*) — The y dimension of a filter kernel. Since PaddlePaddle now supports rectangular filters, the filter's shape can be (filter_size, filter_size_y).
- **num_filters** (*int*) — channel of output data.
- **num_channels** (*int*) — channel of input data.
- **stride** (*int*) — The x dimension of the stride.
- **stride_y** (*int*) — The y dimension of the stride.
- **padding** (*int*) — The x dimension of padding.
- **padding_y** (*int*) — The y dimension of padding.

Returns: A ConvOperator Object.

Return type: ConvOperator

conv_projection

paddle.trainer_config_helpers.layers.**conv_projection**(**args*, ***kwargs*)

ConvProjection with a layer as input. It performs element-wise multiplication with weight.

Different from img_conv_layer and conv_op, conv_projection is an Projection, which can be used in mixed_layer and conat_layer. It use cudnn to implement conv and only support GPU mode.

The example usage is:

```
proj = conv_projection(img=input1,
                       filter_size=3,
                       num_filters=64,
                       num_channels=64)
```

Parameters:
- **input** (*LayerOutput*) — input layer
- **filter_size** (*int*) — The x dimension of a filter kernel.
- **filter_size_y** (*int*) — The y dimension of a filter kernel. Since PaddlePaddle now supports rectangular filters, the filter's shape can be (filter_size, filter_size_y).
- **num_filters** (*int*) — channel of output data.
- **num_channels** (*int*) — channel of input data.
- **stride** (*int*) — The x dimension of the stride.
- **stride_y** (*int*) — The y dimension of the stride.
- **padding** (*int*) — The x dimension of padding.
- **padding_y** (*int*) — The y dimension of padding.
- **groups** (*int*) — The group number.
- **param_attr** (*ParameterAttribute*) — Convolution param attribute. None means default attribute

Returns: A DotMulProjection Object.

Return type: DotMulProjection

conv_shift_layer

paddle.trainer_config_helpers.layers.**conv_shift_layer**(**args*, ***kwargs*)

This layer performs cyclic convolution for two input. For example:
- a[in]: contains M elements.
- b[in]: contains N elements (N should be odd).
- c[out]: contains M elements.

$$c[i] = \sum_{j=-(N-1)/2}^{(N-1)/2} a_{i+j} * b_j$$

In this formular:

- a's index is computed modulo M. When it is negative, then get item from the right side (which is the end of array) to the left.
- b's index is computed modulo N. When it is negative, then get item from the right size (which is the end of array) to the left.

The example usage is:

```
conv_shift = conv_shift_layer(input=[layer1, layer2])
```

Parameters:
- **name** (*basestring*) — layer name
- **a** (*LayerOutput*) — Input layer a.
- **b** (*LayerOutput*) — input layer b
- **layer_attr** (*ExtraLayerAttribute*) — layer's extra attribute.

Returns: LayerOutput object.

Return type: LayerOutput

img_conv_layer

paddle.trainer_config_helpers.layers.**img_conv_layer**(**args*, ***kwargs*)

Convolution layer for image. Paddle only support square input currently and thus input image's width equals height.

The details of convolution layer, please refer UFLDL's convolution .

Convolution Transpose (deconv) layer for image. Paddle only support square input currently and thus input image's width equals height.

The details of convolution transpose layer, please refer to the following explanation and references therein <http://datascience.stackexchange.com/questions/6107/what-are-deconvolutional-layers/>`_ . The num_channel means input image's channel number. It may be 1 or 3 when input is raw pixels of image(mono or RGB), or it may be the previous layer's num_filters * num_group.

There are several group of filter in PaddlePaddle implementation. Each group will process some channel of the inputs. For example, if an input num_channel = 256, group = 4, num_filter=32, the PaddlePaddle will create 32*4 = 128 filters to process inputs. The channels will be split into 4 pieces. First 256/4 = 64 channels will process by first 32 filters. The rest channels will be processed by rest group of filters.

Parameters:
- **name** (*basestring*) — Layer name.
- **input** (*LayerOutput*) — Layer Input.
- **filter_size** (*int|tuple|list*) — The x dimension of a filter kernel. Or input a tuple for two image dimension.
- **filter_size_y** (*int|None*) — The y dimension of a filter kernel. Since PaddlePaddle currently supports rectangular filters, the filter's shape will be (filter_size, filter_size_y).
- **num_filters** — Each filter group's number of filter
- **act** (*BaseActivation*) — Activation type. Default is tanh
- **groups** (*int*) — Group size of filters.
- **stride** (*int|tuple|list*) — The x dimension of the stride. Or input a tuple for two image dimension.

- **stride_y** (*int*) — The y dimension of the stride.
- **padding** (*int|tuple|list*) — The x dimension of the padding. Or input a tuple for two image dimension
- **padding_y** (*int*) — The y dimension of the padding.
- **bias_attr** (*ParameterAttribute|False*) — Convolution bias attribute. None means default bias. False means no bias.
- **num_channels** (*int*) — number of input channels. If None will be set automatically from previous output.
- **param_attr** (*ParameterAttribute*) — Convolution param attribute. None means default attribute
- **shared_biases** (*bool*) — Is biases will be shared between filters or not.
- **layer_attr** (*ExtraLayerAttribute*) — Layer Extra Attribute.
- **trans** (*bool*) — true if it is a convTransLayer, false if it is a convLayer

Returns: LayerOutput object.

Return type: LayerOutput

context_projection

paddle.trainer_config_helpers.layers.**context_projection**(**args*, ***kwargs*)

Context Projection.

It just simply reorganizes input sequence, combines "context_len" sequence to one context from context_start. "context_start" will be set to -(context_len - 1) / 2 by default. If context position out of sequence length, padding will be filled as zero if padding_attr = False, otherwise it is trainable.

For example, origin sequence is [A B C D E F G], context len is 3, then after context projection and not set padding_attr, sequence will be [0AB ABC BCD CDE DEF EFG FG0].

Parameters:

- **input** (*LayerOutput*) — Input Sequence.
- **context_len** (*int*) — context length.
- **context_start** (*int*) — context start position. Default is -(context_len - 1)/2
- **padding_attr** (*bool|ParameterAttribute*) — Padding Parameter Attribute. If false, it means padding always be zero. Otherwise Padding is learnable, and parameter attribute is set by this parameter.

Returns: Projection

Return type: Projection

Image Pooling Layer

img_pool_layer

paddle.trainer_config_helpers.layers.**img_pool_layer**(**args*, ***kwargs*)

Image pooling Layer.

The details of pooling layer, please refer ufldl's pooling .

Parameters:
- **padding** (*int*) — pooling padding width.
- **padding_y** (*int|None*) — pooling padding height. It's equal to padding by default.
- **name** (*basestring.*) — name of pooling layer
- **input** (*LayerOutput*) — layer's input
- **pool_size** (*int*) — pooling window width
- **pool_size_y** (*int|None*) — pooling window height. It's eaqual to pool_size by default.
- **num_channels** (*int*) — number of input channel.
- **pool_type** (*BasePoolingType*) — pooling type. MaxPooling or AvgPooling. Default is MaxPooling.
- **stride** (*int*) — stride width of pooling.
- **stride_y** (*int|None*) — stride height of pooling. It is equal to stride by default.
- **layer_attr** (*ExtraLayerAttribute*) — Extra Layer attribute.
- **img_width** (*int|None*) — the width of input feature map. If it is None, the input feature map should be square.

Returns: LayerOutput object.

Return type: LayerOutput

spp_layer

paddle.trainer_config_helpers.layers.**spp_layer**(**args*, ***kwargs*)

Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition. The details please refer to Kaiming He's paper.

Parameters:
- **name** (*basestring*) — layer name.
- **input** (*LayerOutput*) — layer's input.
- **num_channels** (*int*) — number of input channel.
- **pool_type** — Pooling type. MaxPooling or AveragePooling. Default is MaxPooling.
- **pyramid_height** (*int*) — pyramid height.
- **img_width** (*int|None*) — the width of input feature map. If it is None, the input feature map should be square.
- **layer_attr** (*ExtraLayerAttribute*) — Extra Layer Attribute.

Returns: LayerOutput object.

Return type: LayerOutput

maxout_layer

paddle.trainer_config_helpers.layers.**maxout_layer**(**args*, ***kwargs*)

A layer to do max out on conv layer output.

- Input: output of a conv layer.

- Output: feature map size same as input. Channel is (input channel) / groups.

So groups should be larger than 1, and the num of channels should be able to devided by groups.

Please refer to Paper:

- Maxout Networks: http://www.jmlr.org/proceedings/papers/v28/goodfellow13.pdf
- Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks: https://arxiv.org/pdf/1312.6082v4.pdf

The simple usage is:

```
maxout = maxout_layer(input,
                      num_channels=128,
                      groups=4)
```

Parameters:
- **input** (*LayerOutput*) — The input layer.
- **num_channels** (*int|None*) — The channel number of input layer. If None will be set automatically from previous output.
- **groups** (*int*) — The group number of input layer.
- **size_x** (*int|None*) — conv output width. If None will be set automatically from previous output.
- **size_y** (*int|None*) — conv output height. If None will be set automatically from previous output.
- **name** (*None|basestring.*) — The name of this layer, which can not specify.
- **layer_attr** (*ExtraLayerAttribute*) — Extra Layer attribute.

Returns: LayerOutput object.

Return type: LayerOutput

Norm Layer

img_cmrnorm_layer

paddle.trainer_config_helpers.layers.**img_cmrnorm_layer**(**args*, ***kwargs*)

Response normalization across feature maps. The details please refer to Alex's paper.

Parameters:
- **name** (*None|basestring*) — layer name.
- **input** (*LayerOutput*) — layer's input.
- **size** (*int*) — Normalize in number of $size$ feature maps.
- **scale** (*float*) — The hyper-parameter.
- **power** (*float*) — The hyper-parameter.
- **num_channels** — input layer's filers number or channels. If num_channels is None, it will be set automatically.
- **layer_attr** (*ExtraLayerAttribute*) — Extra Layer Attribute.

Returns: LayerOutput object.

Return type: LayerOutput

batch_norm_layer

paddle.trainer_config_helpers.layers.**batch_norm_layer**(**args*, ***kwargs*)

Batch Normalization Layer. The notation of this layer as follow.

x is the input features over a mini-batch.

$$\mu_\beta \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad // \ mini-batch\ mean$$

$$\sigma_\beta^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\beta)^2 \qquad // \ mini-batch\ variance$$

$$\hat{x_i} \leftarrow \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta^2 + \epsilon}} \qquad // \ normalize$$

$$y_i \leftarrow \gamma\hat{x_i} + \beta \qquad // \ scale\ and\ shift$$

The details of batch normalization please refer to this paper.

Parameters:

- **name** (*basestring*) — layer name.
- **input** (*LayerOutput*) — batch normalization input. Better be linear activation. Because there is an activation inside batch_normalization.
- **batch_norm_type** (*None|string, None or "batch_norm" or "cudnn_batch_norm"*) — We have batch_norm and cudnn_batch_norm. batch_norm supports both CPU and GPU. cudnn_batch_norm requires cuDNN version greater or equal to v4 (>=v4). But cudnn_batch_norm is faster and needs less memory than batch_norm. By default (None), we will automaticly select cudnn_batch_norm for GPU and batch_norm for CPU. Otherwise, select batch norm type based on the specified type. If you use cudnn_batch_norm, we suggested you use latest version, such as v5.1.
- **act** (*BaseActivation*) — Activation Type. Better be relu. Because batch normalization will normalize input near zero.
- **num_channels** (*int*) — num of image channels or previous layer's number of filters. None will automatically get from layer's input.
- **bias_attr** (*ParameterAttribute*) — β, better be zero when initialize. So the initial_std=0, initial_mean=1 is best practice.
- **param_attr** (*ParameterAttribute*) — γ, better be one when initialize. So the initial_std=0, initial_mean=1 is best practice.
- **layer_attr** (*ExtraLayerAttribute*) — Extra Layer Attribute.
- **use_global_stats** (*bool|None.*) — whether use moving mean/variance statistics during testing peroid. If None or True, it will use moving mean/variance statistics during testing. If False, it will use the mean and variance of current batch of test data for testing.
- **moving_average_fraction** (*float.*) — Factor used in the moving average computation, referred to as facotr, $runningMean = newMean * (1 - factor) + runningMean * factor$

Returns: LayerOutput object.

Return LayerOutput

type:

sum_to_one_norm_layer

paddle.trainer_config_helpers.layers.**sum_to_one_norm_layer**(*args, **kwargs)

A layer for sum-to-one normalization, which is used in NEURAL TURING MACHINE.

$$out[i] = \frac{in[i]}{\sum_{k=1}^{N} in[k]}$$

where in is a (batchSize x dataDim) input vector, and out is a (batchSize x dataDim) output vector.

The example usage is:

```
sum_to_one_norm = sum_to_one_norm_layer(input=layer)
```

Parameters:
- **input** (*LayerOutput*) — Input layer.
- **name** (*basestring*) — Layer name.
- **layer_attr** (*ExtraLayerAttribute.*) — extra layer attributes.

Returns: LayerOutput object.

Return type: LayerOutput

Recurrent Layers

recurrent_layer

paddle.trainer_config_helpers.layers.**recurrent_layer**(*args, **kwargs)

Simple recurrent unit layer. It is just a fully connect layer through both time and neural network.

For each sequence [start, end] it performs the following computation:

$$out_i = act(in_i) \text{ for } i = start$$
$$out_i = act(in_i + out_{i-1} * W) \text{ for } start < i <= end$$

If reversed is true, the order is reversed:

$$out_i = act(in_i) \text{ for } i = end$$
$$out_i = act(in_i + out_{i+1} * W) \text{ for } start <= i < end$$

Parameters:
- **input** (*LayerOutput*) — Input Layer
- **act** (*BaseActivation*) — activation.
- **bias_attr** (*ParameterAttribute*) — bias attribute.
- **param_attr** (*ParameterAttribute*) — parameter attribute.
- **name** (*basestring*) — name of the layer
- **layer_attr** (*ExtraLayerAttribute*) — Layer Attribute.

Returns: LayerOutput object.

Return type: LayerOutput

lstmemory

paddle.trainer_config_helpers.layers.**lstmemory**(**args*, ***kwargs*)

Long Short-term Memory Cell.

The memory cell was implemented as follow equations.

$$
\begin{aligned}
i_t &= \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) \\
f_t &= \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \\
c_t &= f_t c_{t-1} + i_t tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \\
o_t &= \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o) \\
h_t &= o_t tanh(c_t)
\end{aligned}
$$

NOTE: In PaddlePaddle's implementation, the multiplications $W_{xi}x_t$, $W_{xf}x_t$, $W_{xc}x_t$, $W_{xo}x_t$ are not done in the lstmemory layer, so an additional mixed_layer with full_matrix_projection or a fc_layer must be included in the configuration file to complete the input-to-hidden mappings before lstmemory is called.

NOTE: This is a low level user interface. You can use network.simple_lstm to config a simple plain lstm layer.

Please refer to **Generating Sequences With Recurrent Neural Networks** for more details about LSTM.

Link goes as below.

Parameters:
- **name** (*basestring*) — The lstmemory layer name.
- **input** (*LayerOutput*) — input layer name.
- **reverse** (*bool*) — is sequence process reversed or not.
- **act** (*BaseActivation*) — activation type, TanhActivation by default. h_t
- **gate_act** (*BaseActivation*) — gate activation type, SigmoidActivation by default.
- **state_act** (*BaseActivation*) — state activation type, TanhActivation by default.
- **bias_attr** (*ParameterAttribute|None|False*) — Bias attribute. None means default bias. False means no bias.
- **param_attr** (*ParameterAttribute|None|False*) — Parameter Attribute.
- **layer_attr** (*ExtraLayerAttribute|None*) — Extra Layer attribute

Returns: LayerOutput object.

Return type: LayerOutput

lstm_step_layer

paddle.trainer_config_helpers.layers.**lstm_step_layer**(**args*, ***kwargs*)

LSTM Step Layer. It used in recurrent_group. The lstm equations are shown as follow.

$$
\begin{aligned}
i_t &= \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) \\
f_t &= \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \\
c_t &= f_t c_{t-1} + i_t tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \\
o_t &= \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o) \\
h_t &= o_t tanh(c_t)
\end{aligned}
$$

The input of lstm step is $Wx_t + Wh_{t-1}$, and user should use `mixed_layer` and `full_matrix_projection` to calculate these input vector.

The state of lstm step is c_{t-1}. And lstm step layer will do

$$
\begin{aligned}
i_t &= \sigma(input + W_{ci}c_{t-1} + b_i) \\
&\dots
\end{aligned}
$$

This layer contains two outputs. Default output is h_t. The other output is o_t, which name is 'state' and can use `get_output_layer` to extract this output.

Parameters:
- **name** (*basestring*) — Layer's name.
- **size** (*int*) — Layer's size. NOTE: lstm layer's size, should be equal as `input.size/4`, and should be equal as `state.size`.
- **input** (*LayerOutput*) — input layer. $Wx_t + Wh_{t-1}$
- **state** (*LayerOutput*) — State Layer. c_{t-1}
- **act** (*BaseActivation*) — Activation type. Default is tanh
- **gate_act** (*BaseActivation*) — Gate Activation Type. Default is sigmoid, and should be sigmoid only.
- **state_act** (*BaseActivation*) — State Activation Type. Default is sigmoid, and should be sigmoid only.
- **bias_attr** (*ParameterAttribute*) — Bias Attribute.
- **layer_attr** (*ExtraLayerAttribute*) — layer's extra attribute.

Returns: LayerOutput object.

Return type: LayerOutput

grumemory

`paddle.trainer_config_helpers.layers.`**`grumemory`**(**args*, ***kwargs*)

Gate Recurrent Unit Layer.

The memory cell was implemented as follow equations.

1. update gate z: defines how much of the previous memory to keep around or the unit updates its activations. The update gate is computed by:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z)$$

2. reset gate r: determines how to combine the new input with the previous memory. The reset gate is computed similarly to the update gate:

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$$

3. The candidate activation $\tilde{h}_t$ is computed similarly to that of the traditional recurrent unit:

$$\tilde{h}_t = tanh(Wx_t + U(r_t \odot h_{t-1}) + b)$$

4. The hidden activation h_t of the GRU at time t is a linear interpolation between the previous activation h_{t-1} and the candidate activation $\tilde{h}_t$:

$$h_t = (1 - z_t)h_{t-1} + z_t\tilde{h}_t$$

NOTE: In PaddlePaddle's implementation, the multiplication operations $W_r x_t$, $W_z x_t$ and Wx_t are not computed in gate_recurrent layer. Consequently, an additional mixed_layer with full_matrix_projection or a fc_layer must be included before grumemory is called.

More details can be found by referring to Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.

The simple usage is:

```
gru = grumemory(input)
```

Parameters:
- **name** (*None|basestring*) — The gru layer name.
- **input** (*LayerOutput.*) — input layer.
- **reverse** (*bool*) — Whether sequence process is reversed or not.
- **act** (*BaseActivation*) — activation type, TanhActivation by default. This activation affects the $\tilde{h}_t$.
- **gate_act** (*BaseActivation*) — gate activation type, SigmoidActivation by default. This activation affects the z_t and r_t. It is the σ in the above formula.
- **bias_attr** (*ParameterAttribute|None|False*) — Bias attribute. None means default bias. False means no bias.
- **param_attr** (*ParameterAttribute|None|False*) — Parameter Attribute.
- **layer_attr** (*ExtraLayerAttribute|None*) — Extra Layer attribute
- **size** (*None*) — Stub parameter of size, but actually not used. If set this size will get a warning.

Returns: LayerOutput object.

Return type: LayerOutput

gru_step_layer

paddle.trainer_config_helpers.layers.**gru_step_layer**(**args*, ***kwargs*)

Parameters:
- **input** (*LayerOutput*) —
- **output_mem** —
- **size** —
- **act** —
- **name** —
- **gate_act** —
- **bias_attr** —
- **layer_attr** —

Returns: LayerOutput object.
Return type: LayerOutput

Recurrent Layer Group

memory

paddle.trainer_config_helpers.layers.**memory**(*name*, *size*, *is_seq=False*, *boot_layer=None*, *boot_bias=None*, *boot_bias_active_type=None*, *boot_with_const_id=None*)

The memory layers is a layer cross each time step. Reference this output as previous time step layer `name` 's output.

The default memory is zero in first time step, previous time step's output in the rest time steps.

If boot_bias, the first time step value is this bias and with activation.

If boot_with_const_id, then the first time stop is a IndexSlot, the Arguments.ids()[0] is this `cost_id`.

If boot_layer is not null, the memory is just the boot_layer's output. Set `is_seq` is true boot layer is sequence.

The same name layer in recurrent group will set memory on each time step.

Parameters:
- **name** (*basestring*) — memory's name.
- **size** (*int*) — size of memory.
- **is_seq** (*bool*) — is sequence for boot_layer
- **boot_layer** (*LayerOutput|None*) — boot layer of memory.
- **boot_bias** (*ParameterAttribute|None*) — boot layer's bias
- **boot_bias_active_type** (*BaseActivation*) — boot layer's active type.
- **boot_with_const_id** (*int*) — boot layer's id.

Returns: LayerOutput object which is a memory.
Return type: LayerOutput

recurrent_group

paddle.trainer_config_helpers.layers.**recurrent_group**(**args*, ***kwargs*)

Recurrent layer group is an extremely flexible recurrent unit in PaddlePaddle. As long as the user defines the calculation done within a time step, PaddlePaddle will iterate such a recurrent calculation over sequence input. This is extremely usefull for attention based model, or Neural Turning Machine like models.

The basic usage (time steps) is:

```
def step(input):
    output = fc_layer(input=layer,
                      size=1024,
                      act=LinearActivation(),
                      bias_attr=False)
    return output
```

```
group = recurrent_group(input=layer,
                        step=step)
```

You can see following configs for further usages:

- time steps: lstmemory_group, paddle/gserver/tests/sequence_layer_group.conf, demo/seqToseq/seqToseq_net.py
- sequence steps: paddle/gserver/tests/sequence_nest_layer_group.conf

Parameters:

- **step** (*callable*) — recurrent one time step function.The input of this function is input of the group. The return of this function will be recurrent group's return value.
  The recurrent group scatter a sequence into time steps. And for each time step, will invoke step function, and return a time step result. Then gather each time step of output into layer group's output.
- **name** (*basestring*) — recurrent_group's name.
- **input** (*LayerOutput|StaticInput|SubsequenceInput|list|tuple*) — Input links array.
  LayerOutput will be scattered into time steps. SubsequenceInput will be scattered into sequence steps. StaticInput will be imported to each time step, and doesn't change through time. It's a mechanism to access layer outside step function.
- **reverse** (*bool*) — If reverse is set true, the recurrent unit will process the input sequence in a reverse order.
- **targetInlink** (*LayerOutput|SubsequenceInput*) — the input layer which share info with layer group's output
  Param input specifies multiple input layers. For SubsequenceInput inputs, config should assign one input layer that share info(the number of sentences and the number of words in each sentence) with all layer group's outputs. targetInlink should be one of the layer group's input.

Returns: LayerOutput object.

Return type: LayerOutput

beam_search

paddle.trainer_config_helpers.layers.**beam_search**(**args*, ***kwargs*)

Beam search is a heuristic search algorithm used in sequence generation. It explores a graph by expanding the most promising nodes in a limited set to maintain tractability.

The example usage is:

```
def rnn_step(input):
    last_time_step_output = memory(name='rnn', size=512)
    with mixed_layer(size=512, name='rnn') as simple_rnn:
        simple_rnn += full_matrix_projection(input)
        simple_rnn += last_time_step_output
    return simple_rnn

beam_gen = beam_search(name="decoder",
```

```
                step=rnn_step,
                input=[StaticInput(encoder_last)],
                bos_id=0,
                eos_id=1,
                beam_size=5)
```

Please see the following demo for more details:

- machine translation : demo/seqToseq/translation/gen.conf demo/seqToseq/seqToseq_net.py

Parameters:
- **name** (*base string*) — Name of the recurrent unit that generates sequences.
- **step** (*callable*) —
  A callable function that defines the calculation in a time step, and it is applied to sequences with arbitrary length by sharing a same set of weights.
  You can refer to the first parameter of recurrent_group, or demo/seqToseq/seqToseq_net.py for more details.
- **input** (*list*) — Input data for the recurrent unit
- **bos_id** (*int*) — Index of the start symbol in the dictionary. The start symbol is a special token for NLP task, which indicates the beginning of a sequence. In the generation task, the start symbol is essential, since it is used to initialize the RNN internal state.
- **eos_id** (*int*) — Index of the end symbol in the dictionary. The end symbol is a special token for NLP task, which indicates the end of a sequence. The generation process will stop once the end symbol is generated, or a pre-defined max iteration number is exceeded.
- **max_length** (*int*) — Max generated sequence length.
- **beam_size** (*int*) — Beam search for sequence generation is an iterative search algorithm. To maintain tractability, every iteration only only stores a predetermined number, called the beam_size, of the most promising next words. The greater the beam size, the fewer candidate words are pruned.
- **num_results_per_sample** (*int*) — Number of the generated results per input sequence. This number must always be less than beam size.

Returns: The generated word index.

Return type: LayerOutput

get_output_layer

paddle.trainer_config_helpers.layers.**get_output_layer**(**args*, ***kwargs*)

Get layer's output by name. In PaddlePaddle, a layer might return multiple values, but returns one layer's output. If the user wants to use another output besides the default one, please use get_output_layer first to get the output from input.

Parameters:
- **name** (*basestring*) — Layer's name.
- **input** (*LayerOutput*) — get output layer's input. And this layer should contains multiple outputs.
- **arg_name** (*basestring*) — Output name from input.
- **layer_attr** — Layer's extra attribute.

Returns: LayerOutput object.
Return type: LayerOutput

Mixed Layer

mixed_layer

paddle.trainer_config_helpers.layers.**mixed_layer**(**args*, ***kwargs*)

Mixed Layer. A mixed layer will add all inputs together, then activate. Each inputs is a projection or operator.

There are two styles of usages.

1. When not set inputs parameter, use mixed_layer like this:

```
with mixed_layer(size=256) as m:
    m += full_matrix_projection(input=layer1)
    m += identity_projection(input=layer2)
```

2. You can also set all inputs when invoke mixed_layer as follows:

```
m = mixed_layer(size=256,
                input=[full_matrix_projection(input=layer1),
                       full_matrix_projection(input=layer2)])
```

Parameters:
- **name** (*basestring*) — mixed layer name. Can be referenced by other layer.
- **size** (*int*) — layer size.
- **input** — inputs layer. It is an optional parameter. If set, then this function will just return layer's name.
- **act** (*BaseActivation*) — Activation Type.
- **bias_attr** (*ParameterAttribute or None or bool*) — The Bias Attribute. If no bias, then pass False or something not type of ParameterAttribute. None will get a default Bias.
- **layer_attr** (*ExtraLayerAttribute*) — The extra layer config. Default is None.

Returns: MixedLayerType object can add inputs or layer name.
Return type: MixedLayerType

embedding_layer

paddle.trainer_config_helpers.layers.**embedding_layer**(**args*, ***kwargs*)

Define a embedding Layer.

Parameters:
- **name** (*basestring*) — Name of this embedding layer.
- **input** (*LayerOutput*) — The input layer for this embedding. NOTE: must be Index Data.
- **size** (*int*) — The embedding dimension.

- **param_attr** (*ParameterAttribute|None*) — The embedding parameter attribute. See ParameterAttribute for details.
- **layer_attr** (*ExtraLayerAttribute|None*) — Extra layer Config. Default is None.

Returns: LayerOutput object.

Return type: LayerOutput

scaling_projection

paddle.trainer_config_helpers.layers.**scaling_projection**(**args*, ***kwargs*)

scaling_projection multiplies the input with a scalar parameter and add to the output.

$$out+ = w * in$$

The example usage is:

```
proj = scaling_projection(input=layer)
```

Parameters:
- **input** (*LayerOutput*) — Input Layer.
- **param_attr** (*ParameterAttribute*) — Parameter config, None if use default.

Returns: A ScalingProjection object

Return type: ScalingProjection

dotmul_projection

paddle.trainer_config_helpers.layers.**dotmul_projection**(**args*, ***kwargs*)

DotMulProjection with a layer as input. It performs element-wise multiplication with weight.

$$out.row[i]+ = in.row[i]. * weight$$

where $.*$ means element-wise multiplication.

The example usage is:

```
proj = dotmul_projection(input=layer)
```

Parameters:
- **input** (*LayerOutput*) — Input layer.
- **param_attr** (*ParameterAttribute*) — Parameter config, None if use default.

Returns: A DotMulProjection Object.

Return type: DotMulProjection

dotmul_operator

paddle.trainer_config_helpers.layers.**dotmul_operator**(*a=None*, *b=None*, *scale=1*, ***kwargs*)

DotMulOperator takes two inputs and performs element-wise multiplication:

$$out.row[i] += scale * (x.row[i] .* y.row[i])$$

where .* means element-wise multiplication, and scale is a config scalar, its default value is one.

The example usage is:

```
op = dotmul_operator(x=layer1, y=layer2, scale=0.5)
```

Parameters:
- **a** (*LayerOutput*) — Input layer1
- **b** (*LayerOutput*) — Input layer2
- **scale** (*float*) — config scalar, default value is one.

Returns: A DotMulOperator Object.

Return type: DotMulOperator

full_matrix_projection

paddle.trainer_config_helpers.layers.**full_matrix_projection**(**args*, ***kwargs*)

Full Matrix Projection. It performs full matrix multiplication.

$$out.row[i] += in.row[i] * weight$$

There are two styles of usage.

1. When used in mixed_layer like this, you can only set the input:

```
with mixed_layer(size=100) as m:
    m += full_matrix_projection(input=layer)
```

2. When used as an independant object like this, you must set the size:

```
proj = full_matrix_projection(input=layer,
                              size=100,
                              param_attr=ParamAttr(name='_proj'))
```

Parameters:
- **input** (*LayerOutput*) — input layer
- **size** (*int*) — The parameter size. Means the width of parameter.
- **param_attr** (*ParameterAttribute*) — Parameter config, None if use default.

Returns: A FullMatrixProjection Object.

Return type: FullMatrixProjection

identity_projection

paddle.trainer_config_helpers.layers.**identity_projection**(*input*, *offset=None*)

1. IdentityProjection if offset=None. It performs:

$$out.row[i] += in.row[i]$$

The example usage is:

```
proj = identity_projection(input=layer)
```

2. IdentityOffsetProjection if offset!=None. It likes IdentityProjection, but layer size may be smaller than input size. It select dimesions [offset, offset+layer_size) from input:

$$out.row[i] += in.row[i + \text{offset}]$$

The example usage is:

```
proj = identity_projection(input=layer,
                           offset=10)
```

Note that both of two projections should not have any parameter.

Parameters:
- **input** (*LayerOutput*) — Input Layer.
- **offset** (*int*) — Offset, None if use default.

Returns: A IdentityProjection or IdentityOffsetProjection object

Return type: IdentityProjection or IdentityOffsetProjection

table_projection

paddle.trainer_config_helpers.layers.**table_projection**(**args*, ***kwargs*)

Table Projection. It selects rows from parameter where row_id is in input_ids.

$$out.row[i] += table.row[ids[i]]$$

where out is output, $table$ is parameter, ids is input_ids, and i is row_id.

There are two styles of usage.

1. When used in mixed_layer like this, you can only set the input:

```
with mixed_layer(size=100) as m:
    m += table_projection(input=layer)
```

2. When used as an independant object like this, you must set the size:

```
proj = table_projection(input=layer,
                        size=100,
                        param_attr=ParamAttr(name='_proj'))
```

Parameters:
- **input** (*LayerOutput*) — Input layer, which must contains id fields.
- **size** (*int*) — The parameter size. Means the width of parameter.
- **param_attr** (*ParameterAttribute*) — Parameter config, None if use default.

Returns: A TableProjection Object.

Return TableProjection

type:

trans_full_matrix_projection

paddle.trainer_config_helpers.layers.**trans_full_matrix_projection**(**args*, ***kwargs*)

Different from full_matrix_projection, this projection performs matrix multiplication, using transpose of weight.

$$out.row[i] += in.row[i] * w^{\mathrm{T}}$$

w^{T} means transpose of weight. The simply usage is:

```
proj = trans_full_matrix_projection(input=layer,
                                    size=100,
                                    param_attr=ParamAttr(
                                         name='_proj',
                                         initial_mean=0.0,
                                         initial_std=0.01))
```

Parameters:
- **input** (*LayerOutput*) — input layer
- **size** (*int*) — The parameter size. Means the width of parameter.
- **param_attr** (*ParameterAttribute*) — Parameter config, None if use default.

Returns: A TransposedFullMatrixProjection Object.

Return type: TransposedFullMatrixProjection

Aggregate Layers

pooling_layer

paddle.trainer_config_helpers.layers.**pooling_layer**(**args*, ***kwargs*)

Pooling layer for sequence inputs, not used for Image.

The example usage is:

```
seq_pool = pooling_layer(input=layer,
                         pooling_type=AvgPooling(),
                         agg_level=AggregateLevel.EACH_SEQUENCE)
```

Parameters:
- **agg_level** (*AggregateLevel*) — AggregateLevel.EACH_TIMESTEP or AggregateLevel.EACH_SEQUENCE
- **name** (*basestring*) — layer name.
- **input** (*LayerOutput*) — input layer name.
- **pooling_type** (*BasePoolingType|None*) — Type of pooling, MaxPooling(default), AvgPooling, SumPooling, SquareRootNPooling.
- **bias_attr** (*ParameterAttribute|None|False*) — Bias parameter attribute. False if no bias.
- **layer_attr** (*ExtraLayerAttribute|None*) — The Extra Attributes for layer, such as dropout.

Returns: LayerOutput object.

Return type: LayerType

last_seq

paddle.trainer_config_helpers.layers.**last_seq**(**args*, ***kwargs*)

Get Last Timestamp Activation of a sequence.

Parameters:
- **agg_level** — Aggregated level
- **name** (*basestring*) — Layer name.
- **input** (*LayerOutput*) — Input layer name.
- **layer_attr** (*ExtraLayerAttribute.*) — extra layer attributes.

Returns: LayerOutput object.

Return type: LayerOutput

first_seq

paddle.trainer_config_helpers.layers.**first_seq**(**args*, ***kwargs*)

Get First Timestamp Activation of a sequence.

Parameters:
- **agg_level** — aggregation level
- **name** (*basestring*) — Layer name.
- **input** (*LayerOutput*) — Input layer name.
- **layer_attr** (*ExtraLayerAttribute.*) — extra layer attributes.

Returns: LayerOutput object.

Return type: LayerOutput

concat_layer

paddle.trainer_config_helpers.layers.**concat_layer**(**args*, ***kwargs*)

Concat all input vector into one huge vector. Inputs can be list of LayerOutput or list of projection.

The example usage is:

```
concat = concat_layer(input=[layer1, layer2])
```

Parameters:
- **name** (*basestring*) — Layer name.
- **input** (*list|tuple|collections.Sequence*) — input layers or projections
- **act** (*BaseActivation*) — Activation type.
- **layer_attr** (*ExtraLayerAttribute*) — Extra Layer Attribute.

Returns: LayerOutput object.

Return type: LayerOutput

Reshaping Layers

block_expand_layer

paddle.trainer_config_helpers.layers.**block_expand_layer**(**args*, ***kwargs*)

Expand feature map to minibatch matrix.

- matrix width is: block_y * block_x * num_channels
- matirx height is: outputH * outputW

$$outputH = 1 + (2 * padding_y + imgSizeH - block_y + stride_y - 1)/stride_y$$
$$outputW = 1 + (2 * padding_x + imgSizeW - block_x + stride_x - 1)/stride_x$$

The expand method is the same with ExpandConvLayer, but saved the transposed value. After expanding, output.sequenceStartPositions will store timeline. The number of time steps are outputH * outputW and the dimension of each time step is block_y * block_x * num_channels. This layer can be used after convolution neural network, and before recurrent neural network.

The simple usage is:

```
block_expand = block_expand_layer(input,
                                  num_channels=128,
                                  stride_x=1,
                                  stride_y=1,
                                  block_x=1,
                                  block_x=3)
```

Parameters:
- **input** (*LayerOutput*) — The input layer.
- **num_channels** (*int|None*) — The channel number of input layer.
- **block_x** (*int*) — The width of sub block.
- **block_y** (*int*) — The width of sub block.
- **stride_x** (*int*) — The stride size in horizontal direction.
- **stride_y** (*int*) — The stride size in vertical direction.
- **padding_x** (*int*) — The padding size in horizontal direction.
- **padding_y** (*int*) — The padding size in vertical direction.
- **name** (*None|basestring.*) — The name of this layer, which can not specify.
- **layer_attr** (*ExtraLayerAttribute|None*) — Extra Layer config.

Returns: LayerOutput object.

Return type: LayerOutput

expand_layer

paddle.trainer_config_helpers.layers.**expand_layer**(**args*, ***kwargs*)

A layer for “Expand Dense data or (sequence data where the length of each sequence is one) to sequence data.”

The example usage is:

```
expand = expand_layer(input=layer1,
                      expand_as=layer2,
                      expand_level=ExpandLevel.FROM_TIMESTEP)
```

Parameters:
- **input** (*LayerOutput*) — Input layer
- **expand_as** (*LayerOutput*) — Expand as this layer’s sequence info.

- **name** (*basestring*) — Layer name.
- **bias_attr** (*ParameterAttribute|None|False*) — Bias attribute. None means default bias. False means no bias.
- **expand_level** (*ExpandLevel*) — whether input layer is timestep(default) or sequence.
- **layer_attr** (*ExtraLayerAttribute.*) — extra layer attributes.

Returns: LayerOutput object.

Return type: LayerOutput

repeat_layer

paddle.trainer_config_helpers.layers.**repeat_layer**(**args*, ***kwargs*)

A layer for repeating the input for num_repeats times. This is equivalent to apply concat_layer() with num_repeats same input.

$$y = [x, x, \cdots, x]$$

The example usage is:

```
expand = repeat_layer(layer, 4)
```

Parameters:
- **input** (*LayerOutput*) — Input layer
- **num_repeats** (*int*) — Repeat the input so many times
- **name** (*basestring*) — Layer name.
- **layer_attr** (*ExtraLayerAttribute.*) — extra layer attributes.

Returns: LayerOutput object.

Return type: LayerOutput

Math Layers

addto_layer

paddle.trainer_config_helpers.layers.**addto_layer**(**args*, ***kwargs*)

AddtoLayer.

$$y = f(\sum_{i} x_i + b)$$

where y is output, x is input, b is bias, and f is activation function.

The example usage is:

```
addto = addto_layer(input=[layer1, layer2],
                    act=ReluActivation(),
                    bias_attr=False)
```

This layer just simply add all input layers together, then activate the sum inputs. Each input of this layer should be the same size, which is also the output size of this layer.

There is no weight matrix for each input, because it just a simple add operation. If you want a complicated operation before add, please use mixed_layer.

It is a very good way to set dropout outside the layers. Since not all PaddlePaddle layer support dropout, you can add an add_to layer, set dropout here. Please refer to dropout_layer for details.

Parameters:
- **name** (*basestring*) — Layer name.
- **input** (*LayerOutput|list|tuple*) — Input layers. It could be a LayerOutput or list/tuple of LayerOutput.
- **act** (*BaseActivation*) — Activation Type, default is tanh.
- **bias_attr** (*ParameterAttribute|bool*) — Bias attribute. If False, means no bias. None is default bias.
- **layer_attr** (*ExtraLayerAttribute*) — Extra Layer attribute.

Returns: LayerOutput object.

Return type: LayerOutput

linear_comb_layer

paddle.trainer_config_helpers.layers.**linear_comb_layer**(**args*, ***kwargs*)

A layer for weighted sum of vectors takes two inputs.

- Input: size of weights is M
  size of vectors is M*N
- Output: a vector of size=N

$$z(i) = \sum_{j=0}^{M-1} x(j)y(i+Nj)$$

where $0 \leq i \leq N-1$

Or in the matrix notation:

$$z = x^{\mathrm{T}} Y$$

In this formular:
- x: weights
- y: vectors.
- z: the output.

Note that the above computation is for one sample. Multiple samples are processed in one batch.

The simple usage is:

```
linear_comb = linear_comb_layer(weights=weight, vectors=vectors,
                                size=elem_dim)
```

Parameters:
- **weights** (*LayerOutput*) — The weight layer.
- **vectors** (*LayerOutput*) — The vector layer.

- **size** (*int*) — the dimension of this layer.
- **name** (*basestring*) — The Layer Name.
- **layer_attr** (*ExtraLayerAttribute|None*) — Extra Layer config.

Returns: LayerOutput object.

Return type: LayerOutput

interpolation_layer

paddle.trainer_config_helpers.layers.**interpolation_layer**(**args*, ***kwargs*)

This layer is for linear interpolation with two inputs, which is used in NEURAL TURING MACHINE.

$$y.row[i] = w[i] * x_1.row[i] + (1 - w[i]) * x_2.row[i]$$

where x_1 and x_2 are two (batchSize x dataDim) inputs, w is (batchSize x 1) weight vector, and y is (batchSize x dataDim) output.

The example usage is:

```
interpolation = interpolation_layer(input=[layer1, layer2], weight=layer3)
```

Parameters:
- **input** (*list|tuple*) — Input layer.
- **weight** (*LayerOutput*) — Weight layer.
- **name** (*basestring*) — Layer name.
- **layer_attr** (*ExtraLayerAttribute.*) — extra layer attributes.

Returns: LayerOutput object.

Return type: LayerOutput

bilinear_interp_layer

paddle.trainer_config_helpers.layers.**bilinear_interp_layer**(**args*, ***kwargs*)

This layer is to implement bilinear interpolation on conv layer output.

Please refer to Wikipedia: https://en.wikipedia.org/wiki/Bilinear_interpolation

The simple usage is:

```
bilinear = bilinear_interp_layer(input=layer1, out_size_x=64, out_size_y=64)
```

Parameters:
- **input** (*LayerOutput.*) — A input layer.
- **out_size_x** (*int|None*) — bilinear interpolation output width.
- **out_size_y** (*int|None*) — bilinear interpolation output height.
- **name** (*None|basestring*) — The layer's name, which cna not be specified.
- **layer_attr** (*ExtraLayerAttribute*) — Extra Layer attribute.

Returns: LayerOutput object.

Return type: LayerOutput

power_layer

paddle.trainer_config_helpers.layers.**power_layer**(**args*, ***kwargs*)

This layer applies a power function to a vector element-wise, which is used in NEURAL TURING MACHINE.

$$y = x^w$$

where x is a input vector, w is scalar weight, and y is a output vector.

The example usage is:

```
power = power_layer(input=layer1, weight=layer2)
```

Parameters:
- **input** (*LayerOutput*) — Input layer.
- **weight** (*LayerOutput*) — Weight layer.
- **name** (*basestring*) — Layer name.
- **layer_attr** (*ExtraLayerAttribute.*) — extra layer attributes.

Returns: LayerOutput object.

Return type: LayerOutput

scaling_layer

paddle.trainer_config_helpers.layers.**scaling_layer**(**args*, ***kwargs*)

A layer for multiplying input vector by weight scalar.

$$y = wx$$

where x is size=dataDim input, w is size=1 weight, and y is size=dataDim output.

Note that the above computation is for one sample. Multiple samples are processed in one batch.

The example usage is:

```
scale = scaling_layer(input=layer1, weight=layer2)
```

Parameters:
- **input** (*LayerOutput*) — Input layer.
- **weight** (*LayerOutput*) — Weight layer.
- **name** (*basestring*) — Layer name.
- **layer_attr** (*ExtraLayerAttribute.*) — extra layer attributes.

Returns: LayerOutput object.

Return type: LayerOutput

slope_intercept_layer

paddle.trainer_config_helpers.layers.**slope_intercept_layer**(**args*, ***kwargs*)

This layer for applying a slope and an intercept to the input element-wise. There is no activation and weight.

$$y = slope * x + intercept$$

The simple usage is:

```
scale = slope_intercept_layer(input=input, slope=-1.0, intercept=1.0)
```

Parameters:
- **input** (*LayerOutput*) — The input layer.
- **name** (*basestring*) — The Layer Name.
- **slope** (*float.*) — the scale factor.
- **intercept** (*float.*) — the offset.
- **layer_attr** (*ExtraLayerAttribute|None*) — Extra Layer config.

Returns: LayerOutput object.

Return type: LayerOutput

tensor_layer

paddle.trainer_config_helpers.layers.**tensor_layer**(**args*, ***kwargs*)

This layer performs tensor operation for two input. For example, each sample:

$$y_i = a * W_i * b^{\mathrm{T}}, i = 0, 1, \ldots, K-1$$

In this formular:
- a: the first input contains M elements.
- b: the second input contains N elements.
- y_i: the i–th element of y.
- W_i: the i–th learned weight, shape if [M, N]
- b^{T}: the transpose of b_2.

The simple usage is:

```
tensor = tensor_layer(a=layer1, b=layer2, size=1000)
```

Parameters:
- **name** (*basestring*) — layer name
- **a** (*LayerOutput*) — Input layer a.
- **b** (*LayerOutput*) — input layer b.
- **size** (*int.*) — the layer dimension.
- **act** (*BaseActivation*) — Activation Type. Default is tanh.
- **param_attr** (*ParameterAttribute*) — The Parameter Attribute.
- **bias_attr** (*ParameterAttribute|None|Any*) — The Bias Attribute. If no bias, then pass False or something not type of ParameterAttribute. None will get a default Bias.
- **layer_attr** (*ExtraLayerAttribute|None*) — Extra Layer config.

Returns: LayerOutput object.

Return type: LayerOutput

cos_sim

paddle.trainer_config_helpers.layers.**cos_sim**(**args*, ***kwargs*)

Cosine Similarity Layer. The cosine similarity equation is here.

$$similarity = cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}$$

The size of a is M, size of b is M*N, Similarity will be calculated N times by step M. The output size is N. The scale will be multiplied to similarity.

Note that the above computation is for one sample. Multiple samples are processed in one batch.

Parameters:
- **name** (*basestring*) — layer name
- **a** (*LayerOutput*) — input layer a
- **b** (*LayerOutput*) — input layer b
- **scale** (*float*) — scale for cosine value. default is 5.
- **size** (*int*) — layer size. NOTE size_a * size should equal size_b.
- **layer_attr** (*ExtraLayerAttribute*) — Extra Layer Attribute.

Returns: LayerOutput object.

Return type: LayerOutput

trans_layer

paddle.trainer_config_helpers.layers.**trans_layer**(**args*, ***kwargs*)

A layer for transposition.

$$y = x^{\mathrm{T}}$$

where x is (M x N) input, and y is (N x M) output.

The example usage is:

```
trans = trans_layer(input=layer)
```

Parameters:
- **input** (*LayerOutput*) — Input layer.
- **name** (*basestring*) — Layer name.
- **layer_attr** (*ExtraLayerAttribute.*) — extra layer attributes.

Returns: LayerOutput object.

Return type: LayerOutput

Sampling Layers

maxid_layer

paddle.trainer_config_helpers.layers.**maxid_layer**(**args*, ***kwargs*)

A layer for finding the id which has the maximal value for each sample. The result is stored in output.ids.

The example usage is:

```
maxid = maxid_layer(input=layer)
```

Parameters:
- **input** (*LayerOutput*) — Input layer name.
- **name** (*basestring*) — Layer name.
- **layer_attr** (*ExtraLayerAttribute.*) — extra layer attributes.

Returns: LayerOutput object.

Return type: LayerOutput

sampling_id_layer

paddle.trainer_config_helpers.layers.**sampling_id_layer**(**args*, ***kwargs*)

A layer for sampling id from multinomial distribution from the input layer. Sampling one id for one sample.

The simple usage is:

```
samping_id = sampling_id_layer(input=input)
```

Parameters:
- **input** (*LayerOutput*) — The input layer.
- **name** (*basestring*) — The Layer Name.
- **layer_attr** (*ExtraLayerAttribute|None*) — Extra Layer config.

Returns: LayerOutput object.

Return type: LayerOutput

Cost Layers

cross_entropy

paddle.trainer_config_helpers.layers.**cross_entropy**(**args*, ***kwargs*)

A loss layer for multi class entropy.

```
cost = cross_entropy(input=input_layer,
                     label=label_layer)
```

Parameters:
- **input** (*LayerOutput.*) — The first input layer.
- **label** — The input label.
- **name** (*None|basestring.*) — The name of this layers. It is not necessary.
- **coeff** (*float.*) — The coefficient affects the gradient in the backward.
- **layer_attr** (*ExtraLayerAttribute*) — Extra Layer Attribute.

Returns: LayerOutput object.

Return type: LayerOutput.

cross_entropy_with_selfnorm

paddle.trainer_config_helpers.layers.**cross_entropy_with_selfnorm**(**args*, ***kwargs*)

A loss layer for multi class entropy with selfnorm.

```
cost = cross_entropy_with_selfnorm(input=input_layer,
                                   label=label_layer)
```

Parameters:
- **input** (*LayerOutput.*) — The first input layer.
- **label** — The input label.

- **name** (*None|basestring.*) — The name of this layers. It is not necessary.
- **coeff** (*float.*) — The coefficient affects the gradient in the backward.
- **softmax_selfnorm_alpha** (*float.*) — The scale factor affects the cost.
- **layer_attr** (*ExtraLayerAttribute*) — Extra Layer Attribute.

Returns: LayerOutput object.

Return type: LayerOutput.

multi_binary_label_cross_entropy

paddle.trainer_config_helpers.layers.**multi_binary_label_cross_entropy**(**args*, ***kwargs*)

A loss layer for multi binary label cross entropy.

```
cost = multi_binary_label_cross_entropy(input=input_layer,
                                        label=label_layer)
```

Parameters:
- **input** (*LayerOutput*) — The first input layer.
- **label** — The input label.
- **type** (*basestring*) — The type of cost.
- **name** (*None|basestring*) — The name of this layers. It is not necessary.
- **coeff** (*float*) — The coefficient affects the gradient in the backward.
- **layer_attr** (*ExtraLayerAttribute*) — Extra Layer Attribute.

Returns: LayerOutput object.

Return type: LayerOutput

huber_cost

paddle.trainer_config_helpers.layers.**huber_cost**(**args*, ***kwargs*)

A loss layer for huber loss.

```
cost = huber_cost(input=input_layer,
                  label=label_layer)
```

Parameters:
- **input** (*LayerOutput.*) — The first input layer.
- **label** — The input label.
- **name** (*None|basestring.*) — The name of this layers. It is not necessary.
- **coeff** (*float.*) — The coefficient affects the gradient in the backward.
- **layer_attr** (*ExtraLayerAttribute*) — Extra Layer Attribute.

Returns: LayerOutput object.

Return type: LayerOutput.

lambda_cost

paddle.trainer_config_helpers.layers.**lambda_cost**(**args*, ***kwargs*)

lambdaCost for lambdaRank LTR approach.

The simple usage:

```
cost = lambda_cost(input=input,
                   score=score,
                   NDCG_num=8,
                   max_sort_size=-1)
```

Parameters:
- **input** (*LayerOutput*) — Samples of the same query should be loaded as sequence.
- **score** — The 2nd input. Score of each sample.
- **NDCG_num** (*int*) — The size of NDCG (Normalized Discounted Cumulative Gain), e.g., 5 for NDCG@5. It must be less than for equal to the minimum size of lists.
- **max_sort_size** (*int*) — The size of partial sorting in calculating gradient. If max_sort_size = –1, then for each list, the algorithm will sort the entire list to get gradient. In other cases, max_sort_size must be greater than or equal to NDCG_num. And if max_sort_size is greater than the size of a list, the algorithm will sort the entire list of get gradient.
- **name** (*None|basestring*) — The name of this layers. It is not necessary.
- **layer_attr** (*ExtraLayerAttribute*) — Extra Layer Attribute.

Returns: LayerOutput object.

Return type: LayerOutput

rank_cost

paddle.trainer_config_helpers.layers.**rank_cost**(**args*, ***kwargs*)

A cost Layer for learning to rank using gradient descent. Details can refer to papers. This layer contains at least three inputs. The weight is an optional argument, which affects the cost.

$$C_{i,j} = -\tilde{P}_{ij} * o_{i,j} + log(1 + e^{o_{i,j}})$$
$$o_{i,j} = o_i - o_j$$
$$\tilde{P}_{i,j} = \{0, 0.5, 1\} \ or \ \{0, 1\}$$

In this formula:
- $C_{i,j}$ is the cross entropy cost.
- $\tilde{P}_{i,j}$ is the label. 1 means positive order and 0 means reverse order.
- o_i and o_j: the left output and right output. Their dimension is one.

The simple usage:

```
cost = rank_cost(left=out_left,
                 right=out_right,
                 label=label)
```

Parameters:
- **left** (*LayerOutput*) — The first input, the size of this layer is 1.
- **right** (*LayerOutput*) — The right input, the size of this layer is 1.
- **label** (*LayerOutput*) — Label is 1 or 0, means positive order and reverse order.
- **weight** (*LayerOutput*) — The weight affects the cost, namely the scale of cost. It is an optional argument.
- **name** (*None|basestring*) — The name of this layers. It is not necessary.
- **coeff** (*float*) — The coefficient affects the gradient in the backward.
- **layer_attr** (*ExtraLayerAttribute*) — Extra Layer Attribute.

Returns: LayerOutput object.

Return type: LayerOutput

crf_layer

paddle.trainer_config_helpers.layers.**crf_layer**(**args*, ***kwargs*)

A layer for calculating the cost of sequential conditional random field model.

The simple usage:

```
crf = crf_layer(input=input,
                label=label,
                size=label_dim)
```

Parameters:
- **input** (*LayerOutput*) — The first input layer is the feature.
- **label** (*LayerOutput*) — The second input layer is label.
- **size** (*int*) — The category number.
- **weight** (*LayerOutput*) — The third layer is "weight" of each sample, which is an optional argument.
- **param_attr** (*ParameterAttribute*) — Parameter attribute. None means default attribute
- **name** (*None|basestring*) — The name of this layers. It is not necessary.
- **layer_attr** (*ExtraLayerAttribute|None*) — Extra Layer config.

Returns: LayerOutput object.

Return type: LayerOutput

crf_decoding_layer

paddle.trainer_config_helpers.layers.**crf_decoding_layer**(**args*, ***kwargs*)

A layer for calculating the decoding sequence of sequential conditional random field model. The decoding sequence is stored in output.ids. If a second input is provided, it is treated as the ground-truth label, and this layer will also calculate error. output.value[i] is 1 for incorrect decoding or 0 for correct decoding.

Parameters:
- **input** (*LayerOutput*) — The first input layer.
- **size** (*int*) — size of this layer.
- **label** (*LayerOutput or None*) — None or ground-truth label.

- **param_attr** (*ParameterAttribute*) — Parameter attribute. None means default attribute
- **name** (*None|basestring*) — The name of this layers. It is not necessary.
- **layer_attr** (*ExtraLayerAttribute|None*) — Extra Layer config.

Returns: LayerOutput object.

Return type: LayerOutput

ctc_layer

paddle.trainer_config_helpers.layers.**ctc_layer**(**args*, ***kwargs*)

Connectionist Temporal Classification (CTC) is designed for temporal classication task. That is, for sequence labeling problems where the alignment between the inputs and the target labels is unknown.

More details can be found by referring to Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks

> **Note:** Considering the 'blank' label needed by CTC, you need to use (num_classes + 1) as the input size. num_classes is the category number. And the 'blank' is the last category index. So the size of 'input' layer, such as fc_layer with softmax activation, should be num_classes + 1. The size of ctc_layer should also be num_classes + 1.

The simple usage:

```
ctc = ctc_layer(input=input,
                label=label,
                size=9055,
                norm_by_times=True)
```

Parameters:
- **input** (*LayerOutput*) — The input layer.
- **label** (*LayerOutput*) — The data layer of label with variable length.
- **size** (*int*) — category numbers + 1.
- **name** (*basestring|None*) — The name of this layer
- **norm_by_times** (*bool*) — Whether to normalization by times. False by default.
- **layer_attr** (*ExtraLayerAttribute|None*) — Extra Layer config.

Returns: LayerOutput object.

Return type: LayerOutput

nce_layer

paddle.trainer_config_helpers.layers.**nce_layer**(**args*, ***kwargs*)

Noise-contrastive estimation. Implements the method in the following paper: A fast and simple algorithm for training neural probabilistic language models.

The example usage is:

```
cost = nce_layer(input=layer1, label=layer2, weight=layer3,
                 num_classes=3, neg_distribution=[0.1,0.3,0.6])
```

Parameters:
- **name** (*basestring*) — layer name
- **input** (*LayerOutput|list|tuple|collections.Sequence*) — input layers. It could be a LayerOutput of list/tuple of LayerOutput.
- **label** (*LayerOutput*) — label layer
- **weight** (*LayerOutput*) — weight layer, can be None(default)
- **num_classes** (*int*) — number of classes.
- **num_neg_samples** (*int*) — number of negative samples. Default is 10.
- **neg_distribution** (*list|tuple|collections.Sequence|None*) — The distribution for generating the random negative labels. A uniform distribution will be used if not provided. If not None, its length must be equal to num_classes.
- **bias_attr** (*ParameterAttribute|None|False*) — Bias parameter attribute. True if no bias.
- **layer_attr** (*ExtraLayerAttribute*) — Extra Layer Attribute.

Returns: layer name.

Return type: LayerOutput

hsigmoid

paddle.trainer_config_helpers.layers.**hsigmoid**(**args*, ***kwargs*)

Organize the classes into a binary tree. At each node, a sigmoid function is used to calculate the probability of belonging to the right branch. This idea is from "F. Morin, Y. Bengio (AISTATS 05): Hierarchical Probabilistic Neural Network Language Model."

The example usage is:

```
cost = hsigmoid(input=[layer1, layer2],
                label=data_layer,
                num_classes=3)
```

Parameters:
- **input** (*LayerOutput|list|tuple*) — Input layers. It could be a LayerOutput or list/tuple of LayerOutput.
- **label** (*LayerOutput*) — Label layer.
- **num_classes** (*int*) — number of classes.
- **name** (*basestring*) — layer name
- **bias_attr** (*ParameterAttribute|False*) — Bias attribute. None means default bias. False means no bias.
- **layer_attr** (*ExtraLayerAttribute*) — Extra Layer Attribute.

Returns: LayerOutput object.

Return type: LayerOutput

sum_cost

paddle.trainer_config_helpers.layers.**sum_cost**(**args*, ***kwargs*)

A loss layer which calculate the sum of the input as loss

```
cost = sum_cost(input=input_layer)
```

Parameters:
- **input** (*LayerOutput.*) — The first input layer.
- **name** (*None|basestring.*) — The name of this layers. It is not necessary.
- **layer_attr** (*ExtraLayerAttribute*) — Extra Layer Attribute.

Returns: LayerOutput object.

Return type: LayerOutput.

Check Layer

eos_layer

paddle.trainer_config_helpers.layers.**eos_layer**(**args*, ***kwargs*)

A layer for checking EOS for each sample: – output_id = (input_id == conf.eos_id)

The result is stored in output_.ids. It is used by recurrent layer group.

The example usage is:

```
eos = eos_layer(input=layer, eos_id=id)
```

Parameters:
- **name** (*basestring*) — Layer name.
- **input** (*LayerOutput*) — Input layer name.
- **eos_id** (*int*) — end id of sequence
- **layer_attr** (*ExtraLayerAttribute.*) — extra layer attributes.

Returns: LayerOutput object.

Return type: LayerOutput