Keras Complex

Oct 16, 2019

Table of Contents

Complex-valued convolutions could provide some interesting results in signal processing-based deep learning. A simple(-ish) idea is including explicit phase information of time series in neural networks. This code enables complex-valued convolution in convolutional neural networks in keras with the TensorFlow backend. This makes the network modular and interoperable with standard keras layers and operations.

CHAPTER 1

Contents

1.1 Introduction

Complex-valued convolutions could provide some interesting results in signal processing-based deep learning. A simple(-ish) idea is including explicit phase information of time series in neural networks. This code enables complex-valued convolution in convolutional neural networks in keras with the TensorFlow backend. This makes the network modular and interoperable with standard keras layers and operations.

1.2 Installation

Installation is as easy as

```
pip install keras-complex
```

but you'll need to install tensorflow in addition using

```
pip install tensorflow-gpu
```

for the GPU version or for the non-GPU version:

```
pip install tensorflow
```

1.3 complexnn

1.3.1 complexnn package

Submodules

complexnn.bn module

complexnn.bn.**ComplexBN**(*input_centred*, *Vrr*, *Vii*, *Vri*, *beta*, *gamma_rr*, *gamma_ri*, *gamma_ii*, *scale=True*, *center=True*, *layernorm=False*, *axis=-1*)

Complex Batch Normalization

Arguments: input_centred – input data Vrr – Real component of covariance matrix V Vii – Imaginary component of covariance matrix V Vri – Non-diagonal component of covariance matrix V beta – Lernable shift parameter beta gamma_rr – Scaling parameter gamma - rr component of 2x2 matrix gamma_ri – Scaling parameter gamma - ri component of 2x2 matrix gamma_ii – Scaling parameter gamma - ii component of 2x2 matrix

Keyword Arguments: scale {bool} {bool} – Standardization of input (default: {True}) center {bool} – Mean-shift correction (default: {True}) layernorm {bool} – Normalization (default: {False}) axis {int} – Axis for Standardization (default: {-1})

Raises: ValueError: Dimonsional mismatch

Returns: Batch-Normalized Input

class complexnn.bn.**ComplexBatchNormalization**(*axis=-1*, *momentum=0.9*, *epsilon=0.0001*, *center=True*, *scale=True*, *beta_initializer='zeros'*, *gamma_diag_initializer='sqrt_init'*, *gamma_off_initializer='zeros'*, *moving_mean_initializer='zeros'*, *moving_variance_initializer='sqrt_init'*, *moving_covariance_initializer='zeros'*, *beta_regularizer=None*, *gamma_diag_regularizer=None*, *gamma_off_regularizer=None*, *beta_constraint=None*, *gamma_diag_constraint=None*, *gamma_off_constraint=None*, ***kwargs*)

Bases: `keras.engine.base_layer.Layer`

Complex version of the real domain Batch normalization layer (Ioffe and Szegedy, 2014). Normalize the activations of the previous complex layer at each batch, i.e. applies a transformation that maintains the mean of a complex unit close to the null vector, the 2 by 2 covariance matrix of a complex unit close to identity and the 2 by 2 relation matrix, also called pseudo-covariance, close to the null matrix. # Arguments

axis: Integer, the axis that should be normalized (typically the features axis). For instance, after a *Conv2D* layer with *data_format="channels_first"*, set *axis=2* in *ComplexBatchNormalization*.

momentum: Momentum for the moving statistics related to the real and imaginary parts.

epsilon: Small float added to each of the variances related to the real and imaginary parts in order to avoid dividing by zero.

center: If True, add offset of *beta* to complex normalized tensor. If False, *beta* is ignored. (beta is formed by real_beta and imag_beta)

scale: If True, multiply by the *gamma* matrix. If False, *gamma* is not used.

beta_initializer: Initializer for the real_beta and the imag_beta weight. gamma_diag_initializer: Initializer for the diagonal elements of the gamma matrix.

which are the variances of the real part and the imaginary part.

gamma_off_initializer: Initializer for the off-diagonal elements of the gamma matrix. moving_mean_initializer: Initializer for the moving means. moving_variance_initializer: Initializer for the moving variances. moving_covariance_initializer: Initializer for the moving covariance of

the real and imaginary parts.

beta_regularizer: Optional regularizer for the beta weights. gamma_regularizer: Optional regularizer for the gamma weights. beta_constraint: Optional constraint for the beta weights. gamma_constraint: Optional constraint for the gamma weights.

 # Input shape Arbitrary. Use the keyword argument *input_shape* (tuple of integers, does not include the samples axis) when using this layer as the first layer in a model.

 # Output shape Same shape as input.

 # References

- [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](https://arxiv.org/abs/1502.03167)

build(*input_shape*)
Creates the layer weights.

Must be implemented on all layers that have weights.

 # Arguments

input_shape: Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.

call(*inputs*, *training=None*)
This is where the layer's logic lives.

 # Arguments inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

 # Returns A tensor or list/tuple of tensors.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

 # Returns Python dictionary.

complexnn.bn.**complex_standardization**(*input_centred*, *Vrr*, *Vii*, *Vri*, *layernorm=False*, *axis=-1*)
Complex Standardization of input

Arguments: input_centred – Input Tensor Vrr – Real component of covariance matrix V Vii – Imaginary component of covariance matrix V Vri – Non-diagonal component of covariance matrix V

Keyword Arguments: layernorm {bool} – Normalization (default: {False}) axis {int} – Axis for Standardization (default: {-1})

Raises: ValueError: Mismatched dimensoins

Returns: Complex standardized input

complexnn.bn.**sanitizedInitGet**(*init*)

complexnn.bn.**sanitizedInitSer**(*init*) `complexnn.bn.sqrt_init(shape, dtype=None)`

complexnn.conv module

conv.py

class complexnn.conv.**ComplexConv**(*rank*, *filters*, *kernel_size*, *strides=1*, *padding='valid'*, *data_format=None*, *dilation_rate=1*, *activation=None*, *use_bias=True*, *normalize_weight=False*, *kernel_initializer='complex'*, *bias_initializer='zeros'*, *gamma_diag_initializer=<function sqrt_init>*, *gamma_off_initializer='zeros'*, *kernel_regularizer=None*, *bias_regularizer=None*, *gamma_diag_regularizer=None*, *gamma_off_regularizer=None*, *activity_regularizer=None*, *kernel_constraint=None*, *bias_constraint=None*, *gamma_diag_constraint=None*, *gamma_off_constraint=None*, *init_criterion='he'*, *seed=None*, *spectral_parametrization=False*, *transposed=False*, *epsilon=1e-07*, ***kwargs*)

Bases: `keras.engine.base_layer.Layer`

Abstract nD complex convolution layer.

This layer creates a complex convolution kernel that is convolved with the layer input to produce a tensor of outputs. If *use_bias* is True, a bias vector is created and added to the outputs. Finally, if *activation* is not *None*, it is applied to the outputs as well.

Arguments:

rank: Integer, the rank of the convolution, e.g., "2" for 2D convolution.

filters: Integer, the dimensionality of the output space, i.e., the number of complex feature maps. It is also the effective number of feature maps for each of the real and imaginary parts. (I.e., the number of complex filters in the convolution) The total effective number of filters is 2 x filters.

kernel_size: An integer or tuple/list of n integers, specifying the dimensions of the convolution window.

strides: An integer or tuple/list of n integers, specifying the strides of the convolution. Specifying any stride value != 1 is incompatible with specifying any *dilation_rate* value != 1.

padding: One of *"valid"* or *"same"* (case-insensitive). data_format: A string, one of *channels_last* (default) or

channels_first. The ordering of the dimensions in the inputs. *channels_last* corresponds to inputs with shape *(batch, …, channels)* while *channels_first* corresponds to inputs with shape *(batch, channels, …)*. It defaults to the *image_data_format* value found in your Keras config file at *~/.keras/keras.json*. If you never set it, then it will be "channels_last".

dilation_rate: An integer or tuple/list of n integers, specifying the dilation rate to use for dilated convolution. Currently, specifying any *dilation_rate* value != 1 is incompatible with specifying any *strides* value != 1.

activation: Activation function to use (see keras.activations). If you don't specify anything, no activation is applied (i.e., "linear" activation: *a(x) = x*).

use_bias: Boolean, whether the layer uses a bias vector. normalize_weight: Boolean, whether the layer normalizes its complex weights before convolving the complex input. The complex normalization performed is similar to the one for the batchnorm. Each of the complex kernels is centred and multiplied by the inverse square root of the covariance matrix. Then a complex multiplication is performed as the normalized weights are multiplied by the complex scaling factor gamma.

kernel_initializer: Initializer for the complex *kernel* weights matrix. By default it is 'complex'. The 'complex_independent' and the usual initializers could also be used. (See keras.initializers and init.py).

bias_initializer: Initializer for the bias vector (see keras.initializers).

kernel_regularizer: Regularizer function applied to the *kernel* weights matrix (see keras.regularizers).

bias_regularizer: Regularizer function applied to the bias vector (see keras.regularizers).

activity_regularizer: Regularizer function applied to the output of the layer (its "activation"). (See keras.regularizers).

kernel_constraint: Constraint function applied to the kernel matrix (see keras.constraints).

bias_constraint: Constraint function applied to the bias vector (see keras.constraints).

spectral_parametrization: Boolean, whether or not to use a spectral parametrization of the parameters.

transposed: Boolean, whether or not to use transposed convolution

`build`(*input_shape*)

`call`(*inputs*, ***kwargs*)
This is where the layer's logic lives.

 # Arguments inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

 # Returns A tensor or list/tuple of tensors.

`compute_output_shape`(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

 # Arguments

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

 # Returns An output shape tuple.

`get_config`()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

 # Returns Python dictionary.

class complexnn.conv.**ComplexConv1D**(*filters*, *kernel_size*, *strides=1*, *padding='valid'*, *dilation_rate=1*, *activation=None*, *use_bias=True*, *kernel_initializer='complex'*, *bias_initializer='zeros'*, *kernel_regularizer=None*, *bias_regularizer=None*, *activity_regularizer=None*, *kernel_constraint=None*, *bias_constraint=None*, *seed=None*, *init_criterion='he'*, *spectral_parametrization=False*, *transposed=False*, ***kwargs*)

Bases: `complexnn.conv.ComplexConv`

1D complex convolution layer. This layer creates a complex convolution kernel that is convolved with a complex input layer over a single complex spatial (or temporal) dimension to produce a complex output tensor. If *use_bias* is True, a bias vector is created and added to the complex output. Finally, if *activation* is not *None*, it is applied each of the real and imaginary parts of the output. When using this layer as the first layer in a model, provide an *input_shape* argument (tuple of integers or *None*, e.g. *(10, 128)* for sequences of 10 vectors of 128-dimensional vectors, or *(None, 128)* for variable-length sequences of 128-dimensional vectors. # Arguments

filters: Integer, the dimensionality of the output space, i.e, the number of complex feature maps. It is also the effective number of feature maps for each of the real and imaginary parts. (i.e. the number of complex filters in the convolution) The total effective number of filters is 2 x filters.

kernel_size: An integer or tuple/list of n integers, specifying the dimensions of the convolution window.

strides: An integer or tuple/list of a single integer, specifying the stride length of the convolution. Specifying any stride value != 1 is incompatible with specifying any *dilation_rate* value != 1.

padding: One of *"valid"*, *"causal"* or *"same"* (case-insensitive). *"causal"* results in causal (dilated) convolutions, e.g. output[t] does not depend on input[t+1:]. Useful when modeling temporal data where the model should not violate the temporal order. See [WaveNet: A Generative Model for Raw Audio, section 2.1] (https://arxiv.org/abs/1609.03499).

dilation_rate: an integer or tuple/list of a single integer, specifying the dilation rate to use for dilated convolution. Currently, specifying any *dilation_rate* value != 1 is incompatible with specifying any *strides* value != 1.

activation: Activation function to use (see keras.activations). If you don't specify anything, no activation is applied (ie. "linear" activation: *a(x) = x*).

use_bias: Boolean, whether the layer uses a bias vector. normalize_weight: Boolean, whether the layer normalizes its complex

weights before convolving the complex input. The complex normalization performed is similar to the one for the batchnorm. Each of the complex kernels are centred and multiplied by the inverse square root of covariance matrix. Then, a complex multiplication is perfromed as the normalized weights are multiplied by the complex scaling factor gamma.

kernel_initializer: Initializer for the complex *kernel* weights matrix.

By default it is 'complex'. The 'complex_independent' and the usual initializers could also be used. (see keras.initializers and init.py).

bias_initializer: Initializer for the bias vector (see keras.initializers).

kernel_regularizer: Regularizer function applied to the *kernel* weights matrix (see keras.regularizers).

bias_regularizer: Regularizer function applied to the bias vector (see keras.regularizers).

activity_regularizer: Regularizer function applied to the output of the layer (its "activation"). (see keras.regularizers).

kernel_constraint: Constraint function applied to the kernel matrix (see keras.constraints).

bias_constraint: Constraint function applied to the bias vector (see keras.constraints).

spectral_parametrization: Whether or not to use a spectral parametrization of the parameters.

transposed: Boolean, whether or not to use transposed convolution

 # Input shape 3D tensor with shape: *(batch_size, steps, input_dim)*

 # Output shape 3D tensor with shape: *(batch_size, new_steps, 2 x filters) steps* value might have changed due to padding or strides.

get_config()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

 # Returns Python dictionary.

class complexnn.conv.**ComplexConv2D**(*filters*, *kernel_size*, *strides=(1, 1)*, *padding='valid'*, *data_format=None*, *dilation_rate=(1, 1)*, *activation=None*, *use_bias=True*, *kernel_initializer='complex'*, *bias_initializer='zeros'*, *kernel_regularizer=None*, *bias_regularizer=None*, *activity_regularizer=None*, *kernel_constraint=None*, *bias_constraint=None*, *seed=None*, *init_criterion='he'*, *spectral_parametrization=False*, *transposed=False*, ***kwargs*)

Bases: complexnn.conv.ComplexConv

2D Complex convolution layer (e.g. spatial convolution over images). This layer creates a complex convolution kernel that is convolved with a complex input layer to produce a complex output tensor. If *use_bias* is True, a complex bias vector is created and added to the outputs. Finally, if *activation* is not *None*, it is applied to both the real and imaginary parts of the output. When using this layer as the first layer in a model, provide the keyword argument *input_shape* (tuple of integers, does not include the sample axis), e.g. *input_shape=(128, 128, 3)* for 128x128 RGB pictures in *data_format="channels_last"*. # Arguments

filters: Integer, the dimensionality of the complex output space (i.e, the number complex feature maps in the convolution). The total effective number of filters or feature maps is 2 x filters.

kernel_size: An integer or tuple/list of 2 integers, specifying the width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.

strides: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any *dilation_rate* value != 1.

padding: one of *"valid"* or *"same"* (case-insensitive). data_format: A string,

one of *channels_last* (default) or *channels_first*. The ordering of the dimensions in the inputs. *channels_last* corresponds to inputs with shape *(batch, height, width, channels)* while *channels_first* corresponds to inputs with shape *(batch, channels, height, width)*. It defaults to the *image_data_format* value found in your Keras config file at *~/.keras/keras.json*. If you never set it, then it will be "channels_last".

dilation_rate: an integer or tuple/list of 2 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any *dilation_rate* value != 1 is incompatible with specifying any stride value != 1.

activation: Activation function to use (see keras.activations). If you don't specify anything, no activation is applied (ie. "linear" activation: *a(x) = x*).

use_bias: Boolean, whether the layer uses a bias vector. normalize_weight: Boolean, whether the layer normalizes its complex

weights before convolving the complex input. The complex normalization performed is similar to the one for the batchnorm. Each of the complex kernels are centred and multiplied by the inverse square root of covariance matrix. Then, a complex multiplication is perfromed as the normalized weights are multiplied by the complex scaling factor gamma.

kernel_initializer: Initializer for the complex *kernel* weights matrix.

By default it is 'complex'. The 'complex_independent' and the usual initializers could also be used. (see keras.initializers and init.py).

bias_initializer: Initializer for the bias vector (see keras.initializers).

kernel_regularizer: Regularizer function applied to the *kernel* weights matrix (see keras.regularizers).

bias_regularizer: Regularizer function applied to the bias vector (see keras.regularizers).

activity_regularizer: Regularizer function applied to the output of the layer (its "activation"). (see keras.regularizers).

kernel_constraint: Constraint function applied to the kernel matrix (see keras.constraints).

bias_constraint: Constraint function applied to the bias vector (see keras.constraints).

spectral_parametrization: Whether or not to use a spectral parametrization of the parameters.

transposed: Boolean, whether or not to use transposed convolution

 # Input shape 4D tensor with shape: *(samples, channels, rows, cols)* if data_format='channels_first' or 4D tensor with shape: *(samples, rows, cols, channels)* if data_format='channels_last'.

 # Output shape 4D tensor with shape: *(samples, 2 x filters, new_rows, new_cols)* if data_format='channels_first' or 4D tensor with shape: *(samples, new_rows, new_cols, 2 x filters)* if data_format='channels_last'. *rows* and *cols* values might have changed due to padding.

get_config()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

 # Returns Python dictionary.

class complexnn.conv.**ComplexConv3D**(*filters*, *kernel_size*, *strides=(1, 1, 1)*, *padding='valid'*, *data_format=None*, *dilation_rate=(1, 1, 1)*, *activation=None*, *use_bias=True*, *kernel_initializer='complex'*, *bias_initializer='zeros'*, *kernel_regularizer=None*, *bias_regularizer=None*, *activity_regularizer=None*, *kernel_constraint=None*, *bias_constraint=None*, *seed=None*, *init_criterion='he'*, *spectral_parametrization=False*, *transposed=False*, ***kwargs*)

Bases: `complexnn.conv.ComplexConv`

3D convolution layer (e.g. spatial convolution over volumes). This layer creates a complex convolution kernel that is convolved with a complex layer input to produce a complex output tensor. If *use_bias* is True, a complex bias vector is created and added to the outputs. Finally, if *activation* is not *None*, it is applied to each of the real and imaginary parts of the output. When using this layer as the first layer in a model, provide the keyword argument *input_shape* (tuple of integers, does not include the sample axis), e.g. *input_shape=(2, 128, 128, 128, 3)* for 128x128x128 volumes with 3 channels, in *data_format="channels_last"*. # Arguments

filters: Integer, the dimensionality of the complex output space (i.e, the number complex feature maps in the convolution). The total effective number of filters or feature maps is 2 x filters.

kernel_size: An integer or tuple/list of 3 integers, specifying the width and height of the 3D convolution window. Can be a single integer to specify the same value for all spatial dimensions.

strides: An integer or tuple/list of 3 integers, specifying the strides of the convolution along each spatial dimension. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any *dilation_rate* value != 1.

padding: one of *"valid"* or *"same"* (case-insensitive). data_format: A string,

one of *channels_last* (default) or *channels_first*. The ordering of the dimensions in the inputs. *channels_last* corresponds to inputs with shape *(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)* while *channels_first* corresponds to inputs with shape *(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)*. It defaults to the *image_data_format* value found in your Keras config file at *~/.keras/keras.json*. If you never set it, then it will be "channels_last".

dilation_rate: an integer or tuple/list of 3 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any *dilation_rate* value != 1 is incompatible with specifying any stride value != 1.

activation: Activation function to use (see keras.activations). If you don't specify anything, no activation is applied (ie. "linear" activation: *a(x) = x*).

use_bias: Boolean, whether the layer uses a bias vector. normalize_weight: Boolean, whether the layer normalizes its complex

weights before convolving the complex input. The complex normalization performed is similar to the one for the batchnorm. Each of the complex kernels are centred and multiplied by the inverse square root of covariance matrix. Then, a complex multiplication is perfromed as the normalized weights are multiplied by the complex scaling factor gamma.

kernel_initializer: Initializer for the complex *kernel* weights matrix. By default it is 'complex'. The 'complex_independent' and the usual initializers could also be used. (see keras.initializers and init.py).

bias_initializer: Initializer for the bias vector (see keras.initializers).

kernel_regularizer: Regularizer function applied to the *kernel* weights matrix (see keras.regularizers).

bias_regularizer: Regularizer function applied to the bias vector (see keras.regularizers).

activity_regularizer: Regularizer function applied to the output of the layer (its "activation"). (see keras.regularizers).

kernel_constraint: Constraint function applied to the kernel matrix (see keras.constraints).

bias_constraint: Constraint function applied to the bias vector (see keras.constraints).

spectral_parametrization: Whether or not to use a spectral parametrization of the parameters.

transposed: Boolean, whether or not to use transposed convolution

 # Input shape 5D tensor with shape: *(samples, channels, conv_dim1, conv_dim2, conv_dim3)* if data_format='channels_first' or 5D tensor with shape: *(samples, conv_dim1, conv_dim2, conv_dim3, channels)* if data_format='channels_last'.

 # Output shape 5D tensor with shape: *(samples, 2 x filters, new_conv_dim1, new_conv_dim2, new_conv_dim3)* if data_format='channels_first' or 5D tensor with shape: *(samples, new_conv_dim1, new_conv_dim2, new_conv_dim3, 2 x filters)* if data_format='channels_last'. *new_conv_dim1*, *new_conv_dim2* and *new_conv_dim3* values might have changed due to padding.

`get_config`()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

 # Returns Python dictionary.

`complexnn.conv.`**`ComplexConvolution1D`**

alias of *`complexnn.conv.ComplexConv1D`*

`complexnn.conv.`**`ComplexConvolution2D`**

alias of *`complexnn.conv.ComplexConv2D`*

`complexnn.conv.`**`ComplexConvolution3D`**

alias of *`complexnn.conv.ComplexConv3D`*

`class` `complexnn.conv.`**`WeightNorm_Conv`**(*gamma_initializer='ones'*, *gamma_regularizer=None*, *gamma_constraint=None*, *epsilon=1e-07*, ***kwargs*)

Bases: `keras.layers.convolutional._Conv`

`build`(*input_shape*)

Creates the layer weights.

Must be implemented on all layers that have weights.

 # Arguments

input_shape: Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.

`call`(*inputs*)

This is where the layer's logic lives.

 # Arguments inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

 # Returns A tensor or list/tuple of tensors.

get_config()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

 # Returns Python dictionary.

complexnn.conv.**conv2d_transpose**(*inputs*, *filter*, *kernel_size=None*, *filters=None*, *strides=(1, 1)*, *padding='SAME'*, *output_padding=None*, *data_format='channels_last'*)

Compatibility layer for K.conv2d_transpose

Take a filter defined for forward convolution and adjusts it for a transposed convolution.

complexnn.conv.**conv_transpose_output_length**(*input_length*, *filter_size*, *padding*, *stride*, *dilation=1*, *output_padding=None*)

Rearrange arguments for compatibility with conv_output_length.

complexnn.conv.**ifft**(*f*)

Stub

complexnn.conv.**ifft2**(*f*)

Stub

complexnn.conv.**sanitizedInitGet**(*init*)

complexnn.conv.**sanitizedInitSer**(*init*)

complexnn.dense module

class complexnn.dense.**ComplexDense**(*units*, *activation=None*, *use_bias=True*, *init_criterion='he'*, *kernel_initializer='complex'*, *bias_initializer='zeros'*, *kernel_regularizer=None*, *bias_regularizer=None*, *activity_regularizer=None*, *kernel_constraint=None*, *bias_constraint=None*, *seed=None*, ***kwargs*)

Bases: `keras.engine.base_layer.Layer`

Regular complex densely-connected NN layer. *Dense* implements the operation: *real_preact = dot(real_input, real_kernel) - dot(imag_input, imag_kernel) imag_preact = dot(real_input, imag_kernel) + dot(imag_input, real_kernel) output = activation(K.concatenate([real_preact, imag_preact]) + bias)* where *activation* is the element-wise activation function passed as the *activation* argument, *kernel* is a weights matrix created by the layer, and *bias* is a bias vector created by the layer (only applicable if *use_bias* is *True*). Note: if the input to the layer has a rank greater than 2, then AN ERROR MESSAGE IS PRINTED. # Arguments

units: Positive integer, dimensionality of each of the real part and the imaginary part. It is actualy the number of complex units.

activation: Activation function to use (see keras.activations). If you don't specify anything, no activation is applied (ie. "linear" activation: *a(x) = x*).

use_bias: Boolean, whether the layer uses a bias vector. kernel_initializer: Initializer for the complex *kernel* weights matrix.

By default it is 'complex'. and the usual initializers could also be used. (see keras.initializers and init.py).

bias_initializer: Initializer for the bias vector (see keras.initializers).

kernel_regularizer: Regularizer function applied to the *kernel* weights matrix (see keras.regularizers).

bias_regularizer: Regularizer function applied to the bias vector (see keras.regularizers).

activity_regularizer: Regularizer function applied to the output of the layer (its "activation"). (see keras.regularizers).

kernel_constraint: Constraint function applied to the kernel matrix (see keras.constraints).

bias_constraint: Constraint function applied to the bias vector (see keras.constraints).

 # Input shape a 2D input with shape *(batch_size, input_dim)*.

 # Output shape For a 2D input with shape *(batch_size, input_dim)*, the output would have shape *(batch_size, units)*.

`build`(*input_shape*)

Creates the layer weights.

Must be implemented on all layers that have weights.

 # Arguments

input_shape: Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.

`call`(*inputs*)

This is where the layer's logic lives.

 # Arguments inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

 # Returns A tensor or list/tuple of tensors.

`compute_output_shape`(*input_shape*)

Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

 # Arguments

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

 # Returns An output shape tuple.

`get_config`()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

 # Returns Python dictionary.

complexnn.fft module

class complexnn.fft.**FFT**(***kwargs*)
Bases: keras.engine.base_layer.Layer

call(*x*, *mask=None*)
This is where the layer's logic lives.

 # Arguments inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

 # Returns A tensor or list/tuple of tensors.

class complexnn.fft.**FFT2**(***kwargs*)
Bases: keras.engine.base_layer.Layer

call(*x*, *mask=None*)
This is where the layer's logic lives.

 # Arguments inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

 # Returns A tensor or list/tuple of tensors.

class complexnn.fft.**IFFT**(***kwargs*)
Bases: keras.engine.base_layer.Layer

call(*x*, *mask=None*)
This is where the layer's logic lives.

 # Arguments inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

 # Returns A tensor or list/tuple of tensors.

class complexnn.fft.**IFFT2**(***kwargs*)
Bases: keras.engine.base_layer.Layer

call(*x*, *mask=None*)
This is where the layer's logic lives.

 # Arguments inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

 # Returns A tensor or list/tuple of tensors.

complexnn.fft.**fft**(*z*)

complexnn.fft.**fft2**(*x*)

complexnn.fft.**ifft**(*z*)

complexnn.fft.**ifft2**(*x*)

complexnn.init module

class complexnn.init.**ComplexIndependentFilters**(*kernel_size*, *input_dim*, *weight_dim*, *nb_filters=None*, *criterion='glorot'*, *seed=None*)
Bases: keras.initializers.Initializer

get_config()

class complexnn.init.**ComplexInit**(*kernel_size*, *input_dim*, *weight_dim*, *nb_filters=None*, *criterion='glorot'*, *seed=None*)
Bases: keras.initializers.Initializer

class complexnn.init.**IndependentFilters**(*kernel_size*, *input_dim*, *weight_dim*, *nb_filters=None*, *criterion='glorot'*, *seed=None*)
Bases: keras.initializers.Initializer

get_config()

class complexnn.init.**SqrtInit**
Bases: keras.initializers.Initializer

complexnn.init.**complex_init**
alias of *complexnn.init.ComplexInit*

complexnn.init.**independent_filters**
alias of *complexnn.init.IndependentFilters*

complexnn.init.**sqrt_init**
alias of *complexnn.init.SqrtInit*

complexnn.norm module

class complexnn.norm.**ComplexLayerNorm**(*epsilon=0.0001*, *axis=-1*, *center=True*, *scale=True*, *beta_initializer='zeros'*, *gamma_diag_initializer=<function sqrt_init>*, *gamma_off_initializer='zeros'*, *beta_regularizer=None*, *gamma_diag_regularizer=None*, *gamma_off_regularizer=None*, *beta_constraint=None*, *gamma_diag_constraint=None*, *gamma_off_constraint=None*, ***kwargs*)
Bases: keras.engine.base_layer.Layer

build(*input_shape*)
Creates the layer weights.

Must be implemented on all layers that have weights.

 # Arguments

input_shape: Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.

call(*inputs*)
This is where the layer's logic lives.

 # Arguments inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

 # Returns A tensor or list/tuple of tensors.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

 # Returns Python dictionary.

class complexnn.norm.**LayerNormalization**(*epsilon=0.0001*, *axis=-1*, *beta_init='zeros'*, *gamma_init='ones'*, *gamma_regularizer=None*, *beta_regularizer=None*, ***kwargs*)

Bases: `keras.engine.base_layer.Layer`

build(*input_shape*)

Creates the layer weights.

Must be implemented on all layers that have weights.

 # Arguments

input_shape: Keras tensor (future input to layer) or list/tuple of Keras tensors to reference for weight shape computations.

call(*x*, *mask=None*)

This is where the layer's logic lives.

 # Arguments inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

 # Returns A tensor or list/tuple of tensors.

get_config()

Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

 # Returns Python dictionary.

complexnn.norm.**layernorm**(*x*, *axis*, *epsilon*, *gamma*, *beta*)

complexnn.pool module

class complexnn.pool.**SpectralPooling1D**(*topf=(0,)*)

Bases: `keras.engine.base_layer.Layer`

call(*x*, *mask=None*)

This is where the layer's logic lives.

 # Arguments inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

 # Returns A tensor or list/tuple of tensors.

class complexnn.pool.**SpectralPooling2D**(***kwargs*)

Bases: `keras.engine.base_layer.Layer`

call(*x*, *mask=None*)

This is where the layer's logic lives.

 # Arguments inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

 # Returns A tensor or list/tuple of tensors.

complexnn.utils module

class complexnn.utils.**GetAbs**(***kwargs*)
Bases: keras.engine.base_layer.Layer

call(*inputs*)
This is where the layer's logic lives.

 # Arguments inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

 # Returns A tensor or list/tuple of tensors.

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

 # Arguments

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

 # Returns An output shape tuple.

class complexnn.utils.**GetImag**(***kwargs*)
Bases: keras.engine.base_layer.Layer

call(*inputs*)
This is where the layer's logic lives.

 # Arguments inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

 # Returns A tensor or list/tuple of tensors.

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

 # Arguments

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

 # Returns An output shape tuple.

class complexnn.utils.**GetReal**(***kwargs*)
Bases: keras.engine.base_layer.Layer

call(*inputs*)
This is where the layer's logic lives.

 # Arguments inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

 # Returns A tensor or list/tuple of tensors.

compute_output_shape(*input_shape*)
Computes the output shape of the layer.

Assumes that the layer will be built to match that input shape provided.

 # Arguments

input_shape: Shape tuple (tuple of integers) or list of shape tuples (one per output tensor of the layer). Shape tuples can include None for free dimensions, instead of an integer.

 # Returns An output shape tuple.

`complexnn.utils.`**`get_abs`**(*x*)

`complexnn.utils.`**`get_imagpart`**(*x*)

`complexnn.utils.`**`get_realpart`**(*x*)

`complexnn.utils.`**`getpart_output_shape`**(*input_shape*)

Module contents

1.4 How to Contribute

1.5 Implementation and Math

Complex convolutional networks provide the benefit of explicitly modelling the phase space of physical systems [TBZ+17]. The complex convolution introduced can be explicitly implemented as convolutions of the real and complex components of both kernels and the data. A complex-valued data matrix in cartesian notation is defined as $\mathbf{M} = M_\Re + iM_\Im$ and equally, the complex-valued convolutional kernel is defined as $\mathbf{K} = K_\Re + iK_\Im$. The individual coefficients $(M_\Re, M_\Im, K_\Re, K_\Im)$ are real-valued matrices, considering vectors are special cases of matrices with one of two dimensions being one.

1.5.1 Complex Convolution Math

The math for complex convolutional networks is similar to real-valued convolutions, with real-valued convolutions being:

$$\int f(y) \cdot g(x-y)\, dy$$

which generalizes to complex-valued function on $\mathbf{R}^d$:

$$(f * g)(x) = \int_{\mathbf{R}^d} f(y)g(x-y)\, dy = \int_{\mathbf{R}^d} f(x-y)g(y)\, dy,$$

in order for the integral to exist, f and g need to decay sufficiently rapidly at infinity [CC-BY-SA Wiki].

1.5.2 Implementation

Solving the convolution of, implemented by [TBZ+17], translated to keras in [DC19]

$$M' = K * M = (M_\Re + iM_\Im) * (K_\Re + iK_\Im),$$

we can apply the distributivity of convolutions to obtain

$$M' = \{M_\Re * K_\Re - M_\Im * K_\Im\} + i\{M_\Re * K_\Im + M_\Im * K_\Re\},$$

where K is the Kernel and M is a data vector.

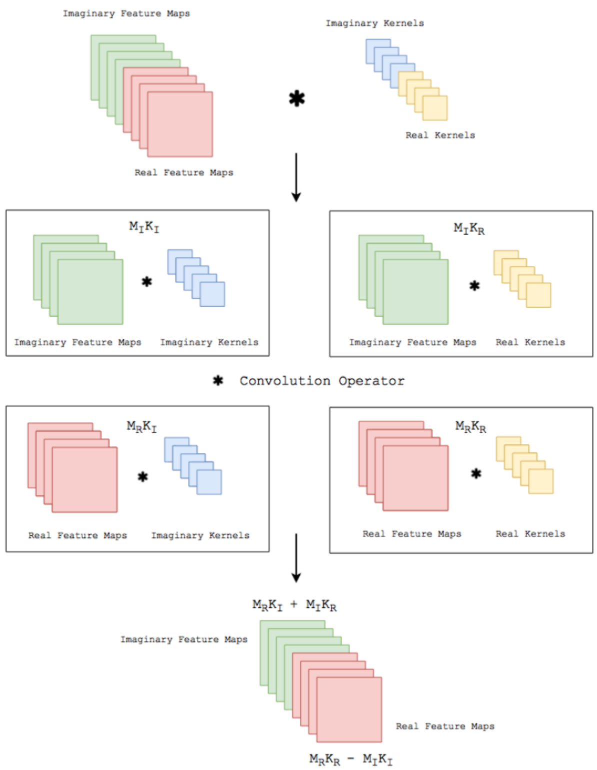

Fig. 1: Complex Convolution implementation (CC-BY [TBZ+17])

1.5.3 Considerations

Complex convolutional neural networks learn by back-propagation. [SSC15] state that the activation functions, as well as the loss function must be complex differentiable (holomorphic). [TBZ+17] suggest that employing complex losses and activation functions is valid for speed, however, refers that [HY12] show that complex-valued networks can be optimized individually with real-valued loss functions and contain piecewise real-valued activations. We reimplement the code [TBZ+17] provides in keras with tensorflow , which provides convenience functions implementing a multitude of real-valued loss functions and activations.

1.6 Citation

Please cite the original work as:

```
@ARTICLE {Trabelsi2017,
    author  = "Chiheb Trabelsi, Olexa Bilaniuk, Ying Zhang, Dmitriy Serdyuk, Sandeep Subramanian, João Felipe Santos, Soroush Mehri, Negar Rostamzadeh, Yoshua Bengio, Christopher J Pal",
    title   = "Deep Complex Networks",
    journal = "arXiv preprint arXiv:1705.09792",
    year    = "2017"
}
```

Cite this software version as:

```
@misc{dramsch2019complex,
    title     = {Complex-Valued Neural Networks in Keras with Tensorflow},
    url       = {https://figshare.com/articles/Complex-Valued_Neural_Networks_in_
↪Keras_with_Tensorflow/9783773/1},
    DOI       = {10.6084/m9.figshare.9783773},
    publisher = {figshare},
    author    = {Dramsch, Jesper S{\"o}ren and Contributors},
    year      = {2019}
}
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Bibliography

[DC19] Jesper Soeren Dramsch and Contributors. Complex-valued neural networks in keras with tensorflow. 2019. URL: https://figshare.com/articles/Complex-Valued_Neural_Networks_in_Keras_with_Tensorflow/9783773/1, doi:10.6084/m9.figshare.9783773.

[DLuthjeC19] Jesper Sören Dramsch, Mikael Lüthje, and Anders Nymark Christensen. Complex-valued neural networks for machine learning on non-stationary physical data. *arXiv preprint arXiv:1905.12321*, 2019.

[HY12] Akira Hirose and Shotaro Yoshida. Generalization characteristics of complex-valued feedforward neural networks in relation to signal coherence. *IEEE Transactions on Neural Networks and Learning Systems*, 2012.

[SSC15] Andy M. Sarroff, Victor Shepardson, and Michael A. Casey. Learning representations using complex-valued nets. *CoRR*, 2015. URL: http://arxiv.org/abs/1511.06351, arXiv:1511.06351.

[TBZ+17] Chiheb Trabelsi, Olexa Bilaniuk, Ying Zhang, Dmitriy Serdyuk, Sandeep Subramanian, João Felipe Santos, Soroush Mehri, Negar Rostamzadeh, Yoshua Bengio, and Christopher J Pal. Deep complex networks. *arXiv preprint arXiv:1705.09792*, 2017.

Python Module Index

C

Index

B

C

F

G

I

L

S

W