Autoregressive Models

PixelCNN and WaveNet

Sun Woo Kim[1]

[1]School of Informatics, Computing, and Engineering
Indiana University Bloomington

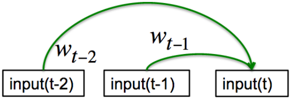
March 22nd, 2018

Outline

Recap (1/2)

- Autoregressive model: value from a sequence is regressed on previous values from that same sequence

$$X_t = w_{t-1}X_{t-1} + w_{t-2}X_{t-2} + \cdots + w_{t-p}X_{t-p} + c$$

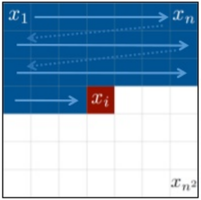


- Objective: Given some training data, use probabilistic density models to generate new images from the same distribution

Figure 1. Image completions sampled from a PixelRNN.

Recap (2/2)

- For each pixel sequentially predict the conditional distribution over the possible pixel values given the scanned context[1]

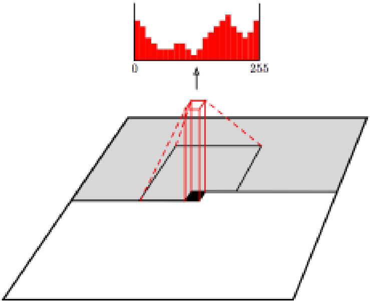


- PixelRNN

[1] https://www.slideshare.net/thinkingfactory/pr12-pixelrnn-jaejun-yoo

Motivation

- Row and Diagonal LSTM layers have long range dependencies in the images
- Also, each state needs to be computed sequentially making the training slow
- Standard convolutional layers can capture a bounded receptive field and compute features for all pixel positions at once



Outline

Architecture

- First Input layer
- Convolutional layer(s)
- Output Convolutional layer(s)
- Softmax layer

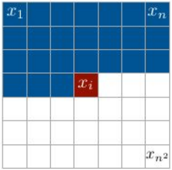
Implementation: First layer

- First layer is a 7x7 masked convolution

```
inputs = tf.placeholder(tf.float32, [None, height, width, channel])

weights_shape = (kernel_d, kernel_d, channel, num_outputs)
weights = tf.get_variable("weights", weights_shape, tf.float32,
                 tf.contrib.layers.xavier_initializer())

maskA = get_mask_A(kernel_d, weights_shape)
weights_masked = weights * tf.constant(maskA, dtype=tf.float32)

conv1 = tf.nn.conv2d(
            inputs, weights_masked, [1, stride, stride, 1], "SAME")
```

Implementation: First layer

- Masks are used in convolutions to avoid seeing future context
- Mask A



Context

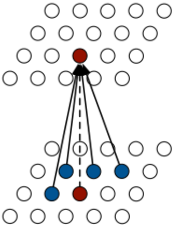
1	1	1	1	1
1	1	1	1	1
1	1	0	0	0
0	0	0	0	0
0	0	0	0	0

Architecture

- First Input layer
- Convolutional layer(s)
- Output Convolutional layer(s)
- Softmax layer

Implementation: Convolutional layers

- Pixel distributions are modeled with convolutional networks
- Use multiple convolutional layers that preserve the spatial resolution; pooling layers are not used
- Much faster to train because convolutions are inherently easier to parallelize



PixelCNN

Implementation: Convolutional layers

- 3x3 Convolution with mask of type B

```
for idx in range(num_layers):
    batch, width, height, channel = l_hid.get_shape().as_list()

    weights_shape = [kernel_d, kernel_d, channel, num_outputs]
    weights = tf.get_variable("weights", weights_shape,
                tf.float32, tf.contrib.layers.xavier_initializer())
    mask = get_mask_B(kernel_d, kernel_d, weights_shape)

    l_hid = tf.nn.conv2d(l_hid, weights * mask, [1,1,1,1],
                padding="SAME", name='outputs')
```

Architecture

- First Input layer
- Convolutional layer(s)
- Output Convolutional layer(s)
- Softmax layer

Implementation: Output Convolutional layers

- The feature map is then passed through a couple of 1x1 convolution layers consisting of ReLU and mask type B

```
batch, width, height, channel = l_hid.get_shape().as_list()
kernel_d = 1

weights_shape = [kernel_d, kernel_d, channel, num_outputs]
weights = tf.get_variable("weights", weights_shape,
            tf.float32, tf.contrib.layers.xavier_initializer())
mask = get_mask_B(kernel_d, kernel_d, weights_shape)

l_hid = tf.n.conv2d(l_hid, weights * mask, [1,1,1,1], padding="SAME")
l_hid = tf.nn.relu(l_hid)
```

Architecture

- First Input layer
- Convolutional layer(s)
- Output Convolutional layer(s)
- Softmax layer

Implementation: Final layer

- 256-way softmax layer

```
batch, width, height, channel = l_hid.get_shape().as_list()
kernel_d = 1
num_outputs = 1

weights_shape = [kernel_d, kernel_d, channel, num_outputs]
weights = tf.get_variable("weights", weights_shape,
            tf.float32, tf.contrib.layers.xavier_initializer())
mask = get_mask_B(kernel_d, kernel_d, weights_shape)

l_hid = tf.n.conv2d(l_hid, weights * mask, [1,1,1,1], padding="SAME")
output = tf.nn.sigmoid(l_hid)
```

Outline

Improvements: Worse than PixelRNN

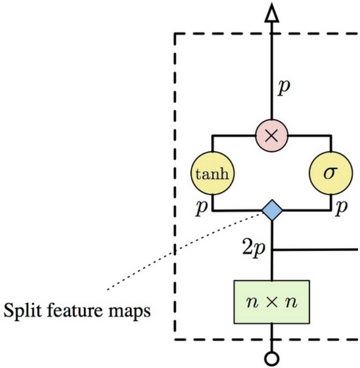
- PixelRNNs which use LSTM layers instead of convolutional stacks outperform PixelCNNs
- Recurrent connections in LSTM allow every layer in the network to access the entire neighborhood of previous pixels
- The region available to pixelCNN grows linearly in depth of the convolutional stack

- Solution: Use sufficiently many layers

Improvements: Worse than PixelRNN

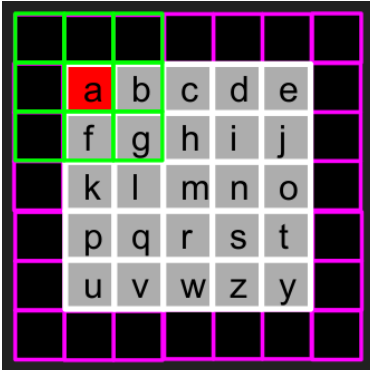
- PixelRNNs contain multiplicative units in the form of LSTM gates that help it model more complex interactions
- Solution: replace ReLU between the masked convolutions with a gated activation unit

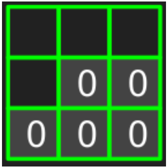
$$\mathbf{y} = tanh(W_{k.f} * \mathbf{x}) \odot \sigma(W_{k,g} * \mathbf{x})$$



Improvements: Blind Spot Problem

- Masked convolutional architecture[2]

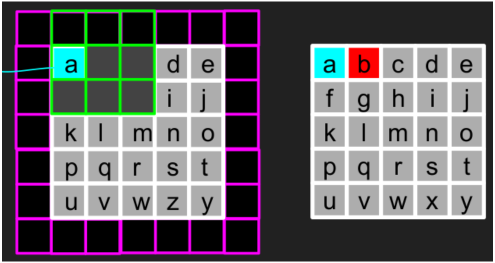




[2] https://towardsdatascience.com/blind-spot-problem-in-pixelcnn-8c71592a14a

Improvements: Blind Spot Problem

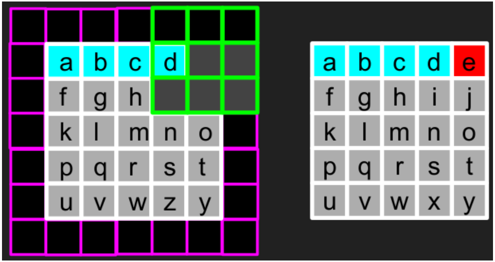
- Value of b depends on a, the previous predicted value[3]



[3] https://towardsdatascience.com/blind-spot-problem-in-pixelcnn-8c71592a14a

Improvements: Blind Spot Problem

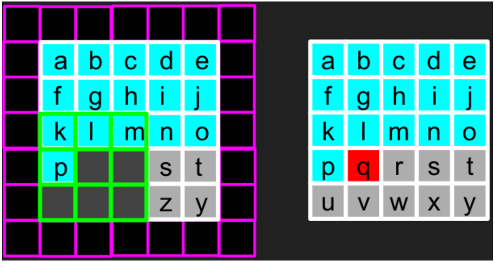
- Pixel e includes d in its filter, which depends on c and so on[4]
- Thus, e directly or indirectly depends on a, b, c, d.



[4] https://towardsdatascience.com/blind-spot-problem-in-pixelcnn-8c71592a14a

Improvements: Blind Spot Problem

- Pixel q depends on k, l, m, p[5]
- k, l, m, p depend on f, g, h, i
- f, g, h, i depend on a through e



[5] https://towardsdatascience.com/blind-spot-problem-in-pixelcnn-8c71592a14a

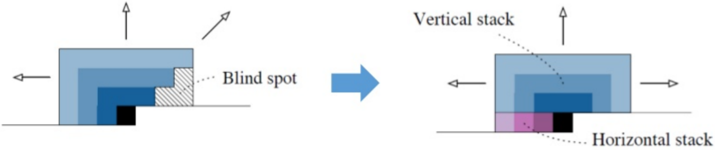
Improvements: Blind Spot Problem

- Pixel q depends on all its previous pixels except n, o, j; these pixels are not used in the prediction[6]

[6] https://towardsdatascience.com/blind-spot-problem-in-pixelcnn-8c71592a14a

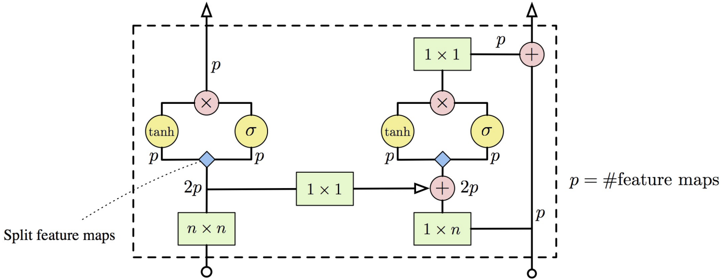
Improvements: Blind Spot Problem

- Blind spot problem: pixels predicted using PixelCNN, are not dependent on all previous pixels
- Solution: combine two convolutional stacks



  - Vertical stack conditions on all the rows above
  - Horizontal stack conditions on the current row so far

Improvements: Blind Spot Problem

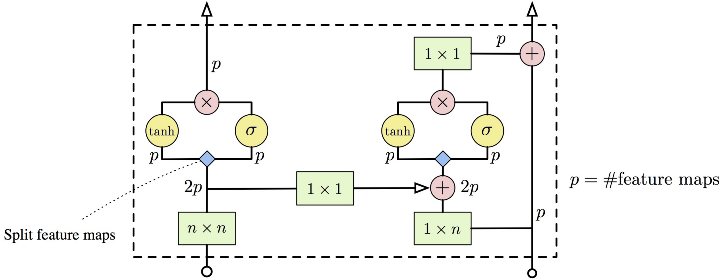


- Residual connection in the horizontal stack

Architecture

- First Input layer
- Gated Convolutional layer(s)
- Output Convolutional layer(s)
- Softmax layer

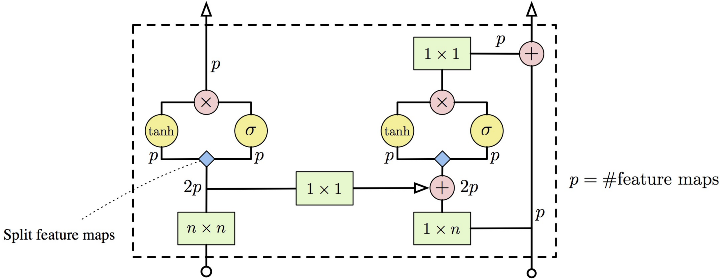
Implementation: Gated CNN



- Vertical: $n \times n$ convolution

```
vert_nxn = conv(v_inputs, p * 2,
                kernel_shape, 'V',
                num_channels)
```

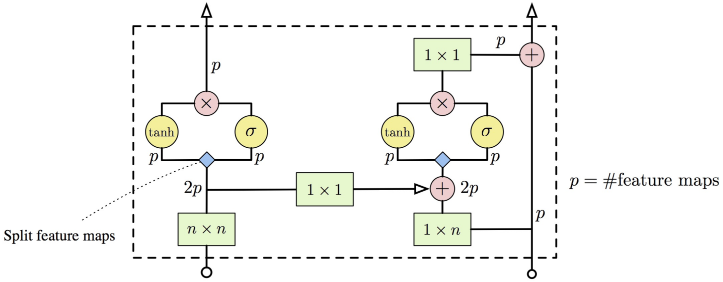
Implementation: Gated CNN



- Vertical: gated activation

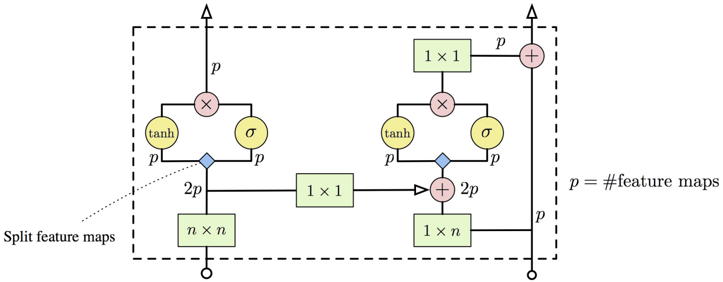
```
vert_gated_out = _gated_activation_unit(
                        vert_nxn, kernel_shape,
                        'V', num_channels)
```

Implementation: Gated CNN




```python
def gated_act(inputs, kernel_shape, mask_type, num_channels):
    conv_out = conv(inputs, p2, kernel_shape,
                    mask_type, num_channels)
    left, right = tf.split(conv_out, 2, 3)
    return tf.tanh(left) * tf.sigmoid(right)
```

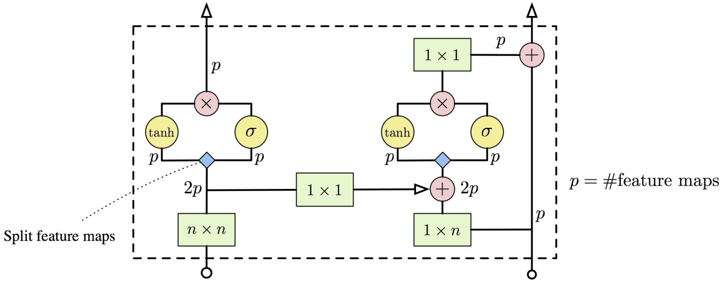
Implementation: Gated CNN



- Vertical: 1 x 1 convolution

```
vert_1x1 = conv(vert_nxn, p2, [1, 1], 'V', num_channels)
```

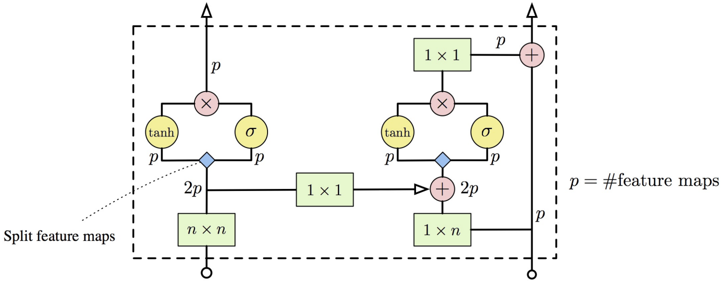
Implementation: Gated CNN



- Horizontal: 1 x n convolution

```
horiz_1xn = conv(h_inputs, p2, [1,3], 'B', num_channels)
```

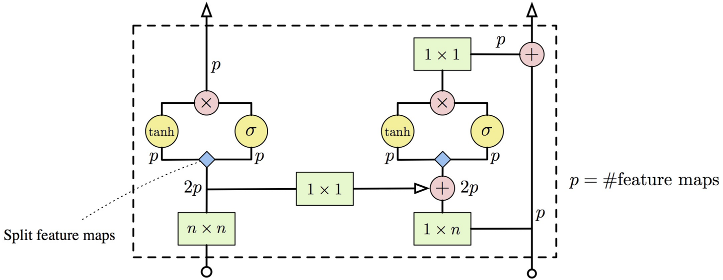
Implementation: Gated CNN



- Horizontal: before gated activation

```
horiz_gated_in = vert_1x1 + horiz_1xn
```

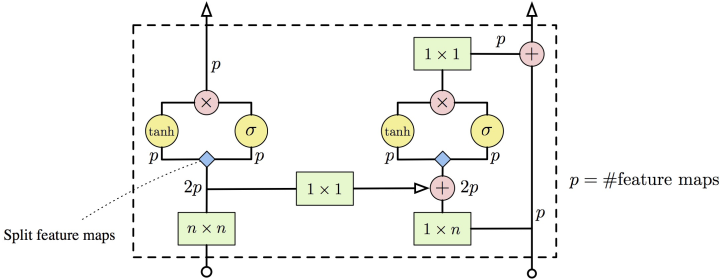
Implementation: Gated CNN



- Horizontal: gated activation

```
horiz_gated_out = _gated_activation_unit(
                    horiz_gated_in, kernel_shape,
                    'B', num_channels)
```

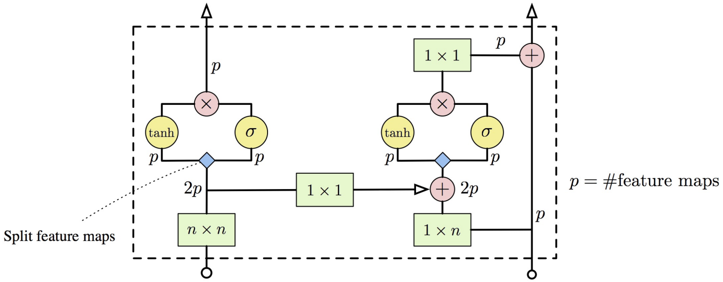
Implementation: Gated CNN



- Horizontal: 1 x 1 convolution

```
horiz_1x1 = conv(horiz_gated_out, p,
                 [1,1], 'B', num_channels)
```

Implementation: Gated CNN



- Horizontal: residual connection

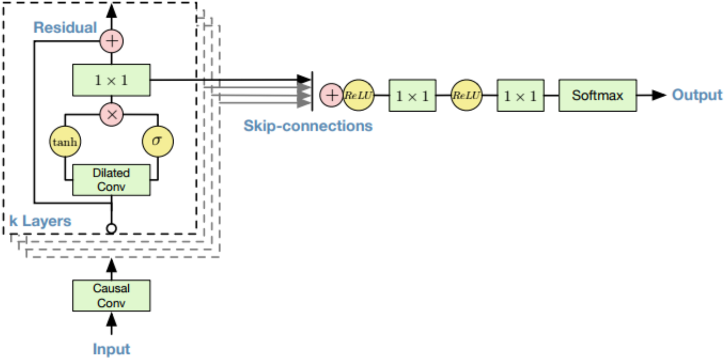
```
horiz_outputs = horiz_1x1 + h_inputs
```

PixelCNN Architecture

- First Input layer
- Gated Convolutional layer(s)
- Output Convolutional layer(s)
- Softmax layer

Outline

Architecture



Outline

Preprocessing

One Hot Encoding

- Transform into one hot vector for softmax output

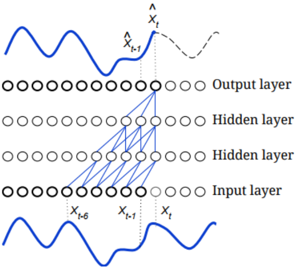
```
one_hot_encoded = tf.one_hot(
                    encoded_input,
                    depth=QUANTIZATION_CHANNELS,
                    dtype=tf.float32)
```

Reshape for batch processing

```
encoded = tf.reshape(
            one_hot_encoded,
            [BATCH_SIZE, -1, QUANTIZATION_CHANNELS])
```

Preprocessing

Preserve Causality in First Layer

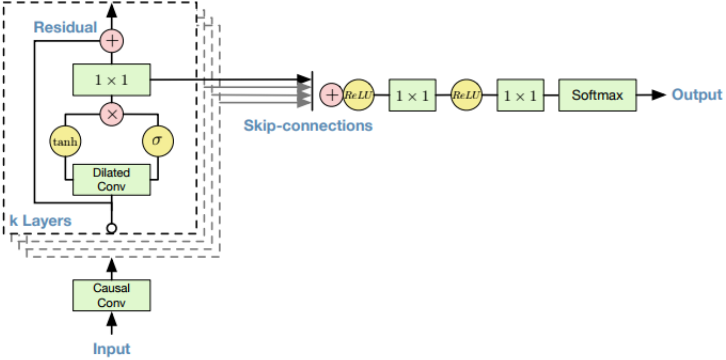


- Break information flow from sample being predicted and your network

```
network_input_width = tf.shape(encoded)[1] - 1
network_input = tf.slice(encoded, [0, 0, 0],
                         [-1, network_input_width, -1])
```

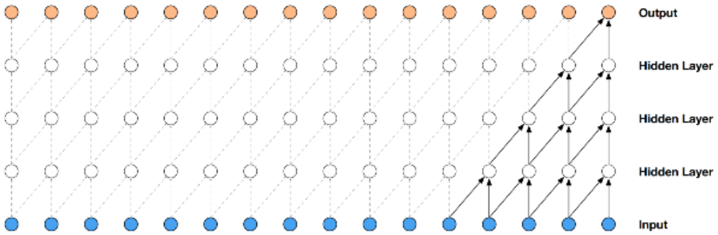
Outline

Initial Causal Convolution



Causal Convolutions

- Ordering cannot be violated
- Fast to train but require many layers to increase the receptive field



Causal Convolution

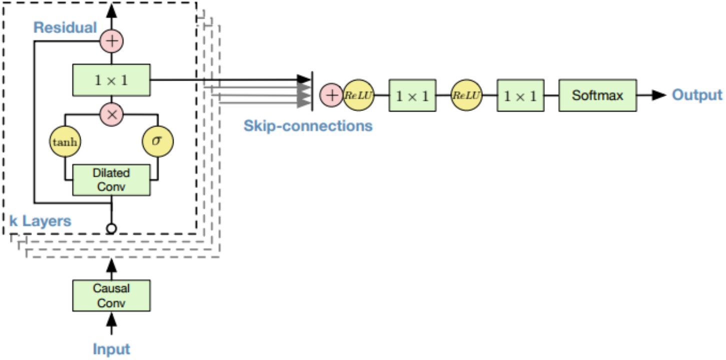
- Input: Encoded Audio Input
- 1x1 Regular Convolution
  - Weights: Initial Causal Filter
  - Output: 1D Convoluted Output

```
w_init = tf.contrib.layers.xavier_initializer_conv2d()
w_shape = [initial_filter_width, initial_channels,
           RESIDUAL_CHANNELS]
weights_filter = tf.Variable(w_init(shape=w_shape))

current_layer_causal = tf.nn.conv1d(
                        current_layer, weights_filter,
                        stride=1, padding='VALID')
```

Outline

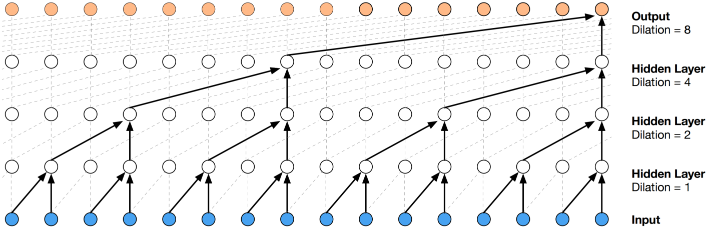
Architecture

Dilated Convolution



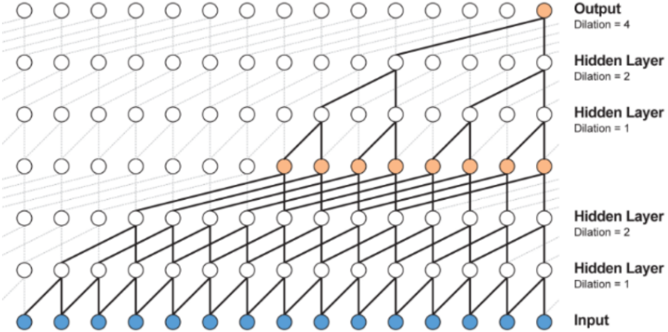
Dilated Causal Convolutions

- Skip input values with a certain step
- Operate on a coarser scale



Stacked Dilated Causal Convolutions

- Dilation is doubled and repeated (1, 2, 4, 1, 2, 4, 1, ...)
- Large receptive fields with few layers; computationally efficient
- Link

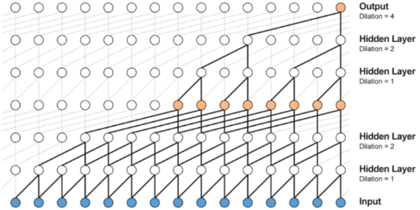


Dilated Causal Convolution

- Input: Output from previous layer
- Dilated Convolution
  - Weights:
    - Weights Filter
    - Weights Gate
  - Output: 2 Dilated Convoluted Outputs

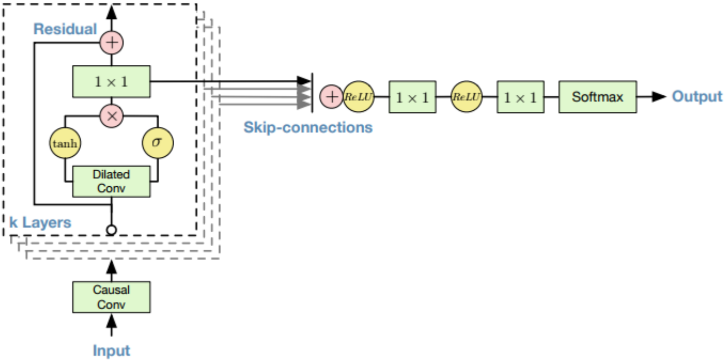
```
weights_shape = [FILTER_WIDTH, RESIDUAL_CHANNELS, DILATION_CHANNELS]
weights_filter = tf.Variable(w_init(shape=weights_shape))
weights_gate = tf.Variable(w_init(shape=weights_shape))
conv_filter = dilated_causal_conv(input_batch, weights_filter, dilation)
conv_gate = dilated_causal_conv(input_batch, weights_gate, dilation)
```

Stacked Dilated Causal Convolution




```python
def dilated_causal_conv(input_batch, weights_filter, dilation):
    shape = tf.shape(input_batch)
    pad_elements = dilation - 1 - (shape[1] + dilation - 1) % dilation
    padded = tf.pad(input_batch, [[0, 0], [0, pad_elements], [0, 0]])
    reshaped = tf.reshape(padded, [-1, dilation, shape[2]])
    transposed = tf.transpose(reshaped, perm=[1, 0, 2])

    conv = tf.nn.conv1d(transposed, weights_filter, stride=1, padding='VALID')

    shape = tf.shape(conv)
    transposed = tf.transpose(conv, perm=[1, 0, 2])
    restored = tf.reshape(transposed, [1, -1, shape[2]])

    return restored
```

Architecture

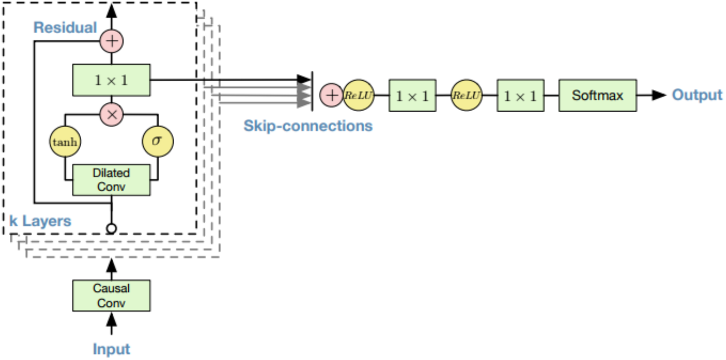


- Activations:

```
out = tf.tanh(conv_filter) * tf.sigmoid(conv_gate)
```

Outline

Architecture

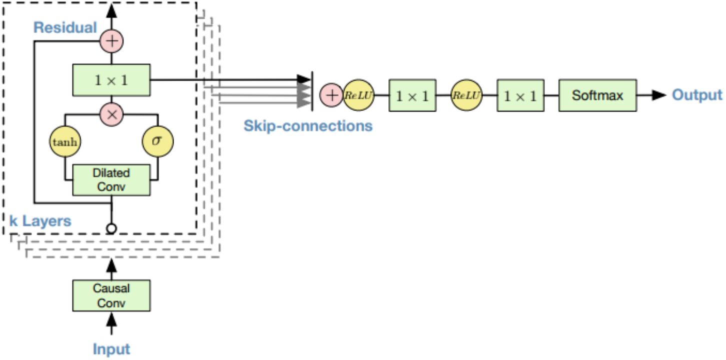


Residual Connections

- Input: Output of Dilated Convolution with Activations
- 1x1 Regular Convolution
  - Weights: Dense Filter
  - Output: 1D Convoluted Output (input to next layer)

```
weights_dense = tf.Variable(w_init(
                    shape=[1, DILATION_CHANNELS, RESIDUAL_CHANNELS]),
                    name='Wd')
transformed = tf.nn.conv1d(out, weights_dense, stride=1, padding="SAME")
current_layer = input_batch + transformed
```

Architecture



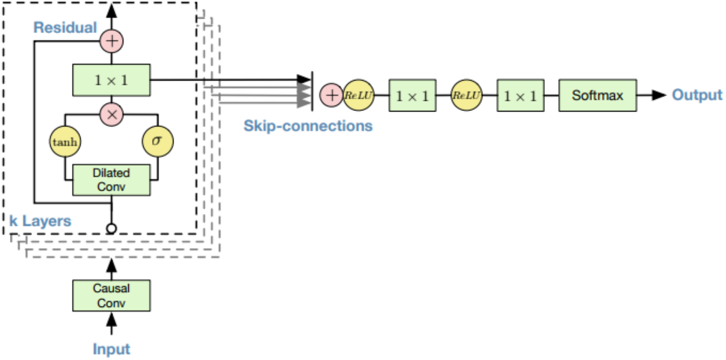
Skip Connections

- Input: Output of Dilated Convolution with Activations
- 1x1 Regular Convolution
  - Weights: Skip Filter
  - Output: List of # layers of outputs

```
weights_shape = [1, DILATION_CHANNELS, RESIDUAL_CHANNELS]
weights_skip = tf.Variable(w_init(shape=weights_shape))
skip_contribution = tf.nn.conv1d(
                        output, weights_skip, stride=1,
                        padding="SAME", name="skip")
```

Outline

Architecture



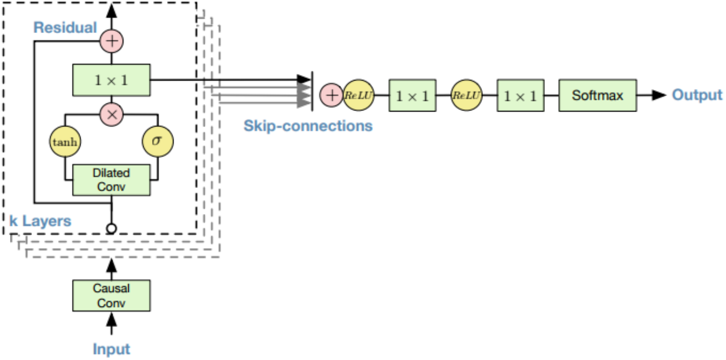
Postprocessing

- Sum the outputs
- ReLU → 1x1 Convolution → ReLU → 1x1 Convolution

```
w1 = tf.Variable(w_init(
                shape=[1, SKIP_CHANNELS, SKIP_CHANNELS]),
                 name='postprocess1')
w2 = tf.Variable(w_init(
                shape=[1, SKIP_CHANNELS, QUANTIZATION_CHANNELS]),
                name='postprocess2')

total = sum(outputs)
transformed1 = tf.nn.relu(total)
conv1 = tf.nn.conv1d(transformed1, w1, stride=1, padding="SAME")
transformed2 = tf.nn.relu(conv1)
conv2 = tf.nn.conv1d(transformed2, w2, stride=1, padding="SAME")
```

Outline

Architecture



Output and Loss Function

- Softmax Output

```
prediction = tf.reshape(raw_output, [-1, QUANTIZATION_CHANNELS])
target_output = tf.reshape(encoded, [BATCH_SIZE, -1, QUANTIZATION_CHANNELS])

loss = tf.nn.softmax_cross_entropy_with_logits(logits=prediction, labels=target_output)
reduced_loss = tf.reduce_mean(loss)

# Set up training
optimizer = tf.train.AdamOptimizer(learning_rate=LEARNING_RATE)
optim = optimizer.minimize(reduced_loss, var_list=tf.trainable_variables())
```

Results

- Link