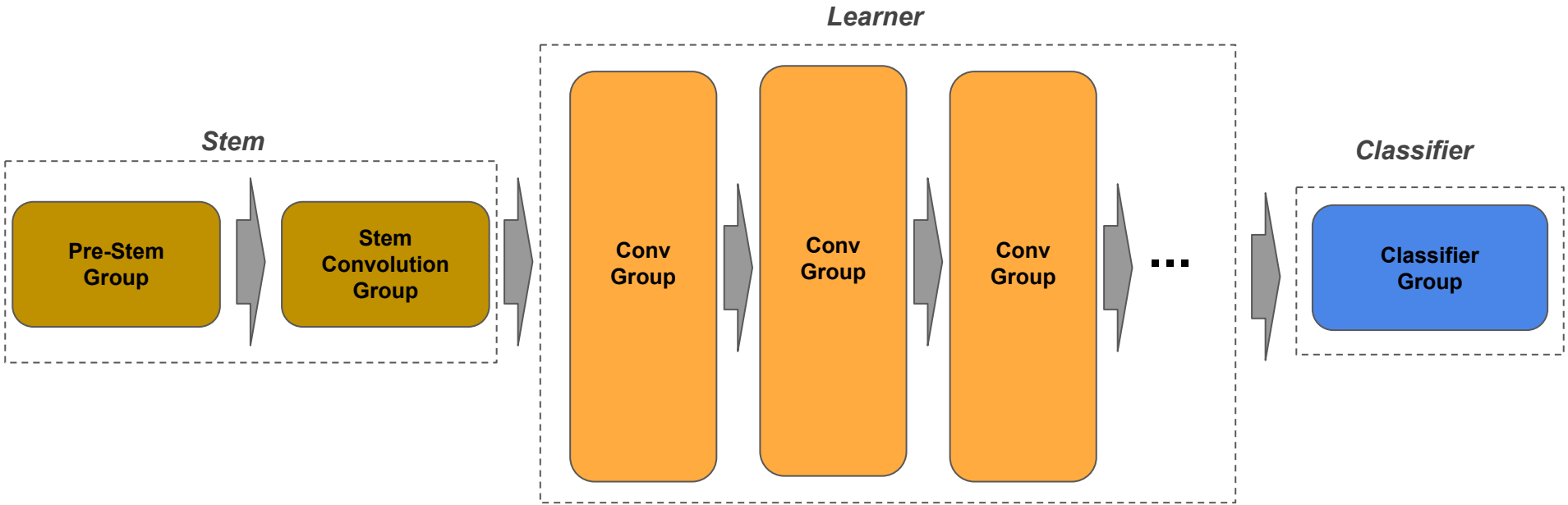
Modern Convolutional Neural Network Architectures

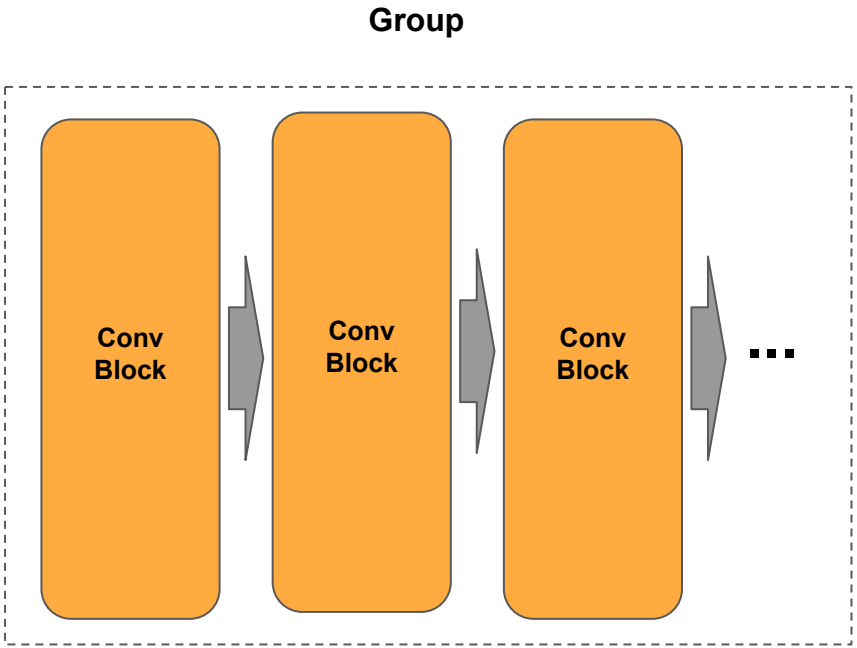
Andrew Ferlitsch
Google Cloud AI/Developer Relations

Repo: https://github.com/GoogleCloudPlatform/keras-idiomatic-programmer/tree/master/workshops/Modern_CNN

Macro Architecture



Micro Architecture



Parameters

Meta-Parameters - These are the parameters for configuring macro-architecture.

Hyper-parameters - These are the parameters for training the model.

Parameters - These are the parameters the model will learn during training.

Macro Architecture

```
def stem(inputs):
    ...

def learner(inputs, **metaparameters):
    ...

def classifier(inputs, n_classes):
    ...

inputs = Input(shape=...)

layers = stem(inputs)

layers = learner(layers, metaparameters=...)

outputs = classifier(layers, n_classes)

model = Model(inputs, outputs)
```

Procedural Style (Idiomatic) of coding the macro architecture of a model in TF.Keras.

Macro Architecture

```
class MyModel():
    def init(self, input_shape, n_classes, **metaparameters):
        inputs = Input(shape=input_shape)
        layers = stem(inputs)
        layers = learner(layers, metaparameters)
        outputs = classifier(layers, n_classes)
        model = Model(inputs, outputs)

    def stem(self, inputs):
        ...

    def learner(self, inputs, **metaparameters):
        ...

    def classifier(self, inputs, n_classes):
        ...
```

OOP Style (Composable) of coding a model in TF.Keras.

Micro Architecture

```
def learner(inputs, **metaparameters):
    ...

    for group_params in metaparameters['groups']:
      inputs = group(inputs, group_parameters)

def group(inputs, **metaparameters):

    for block_params in metaparameters['n_blocks']:
      inputs = block(inputs, block_parameters)

def block(inputs, **metaparameters):
    ...

metaparameters = {
'groups' :[ { n_blocks: 4, filters: 32 }, {n_blocks: 8, filters:64} ] }
```

Procedural Style (Idiomatic) of coding the micro architecture of a model in TF.Keras.

Micro Architecture

```
class MyModel():
    metaparameters = {
    'groups' :[ { n_blocks: 4, filters: 32 }, {n_blocks: 8, filters:64}
    ] }

   def learner(self, inputs, **metaparameters):
        ...

        for group_params in metaparameters['groups']:
            inputs = group(inputs, group_parameters)

   @staticmethod
   def group(inputs, **metaparameters):

        for block_params in metaparameters['n_blocks']:
            inputs = block(inputs, block_parameters)

   @staticmethod
    def block(inputs, **metaparameters):
        ...
```

OOP Style (Composable) of coding the micro architecture of a model in TF.Keras.

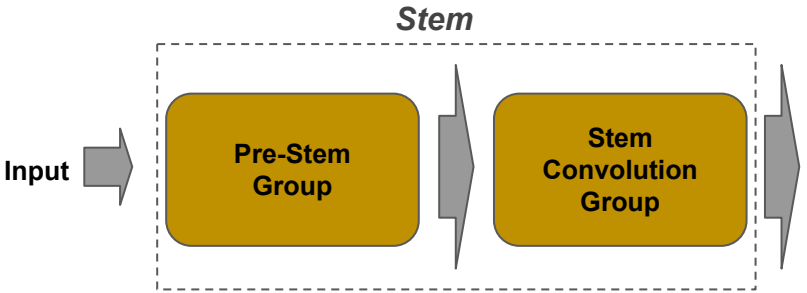
@staticmethod provides means to tear off buildable micro components that afe configured by metaparameters (factory design pattern).

Stem

Data transformations (T in ETL) of raw data:

Image preprocessing
Image augmentation

post processing for prediction.

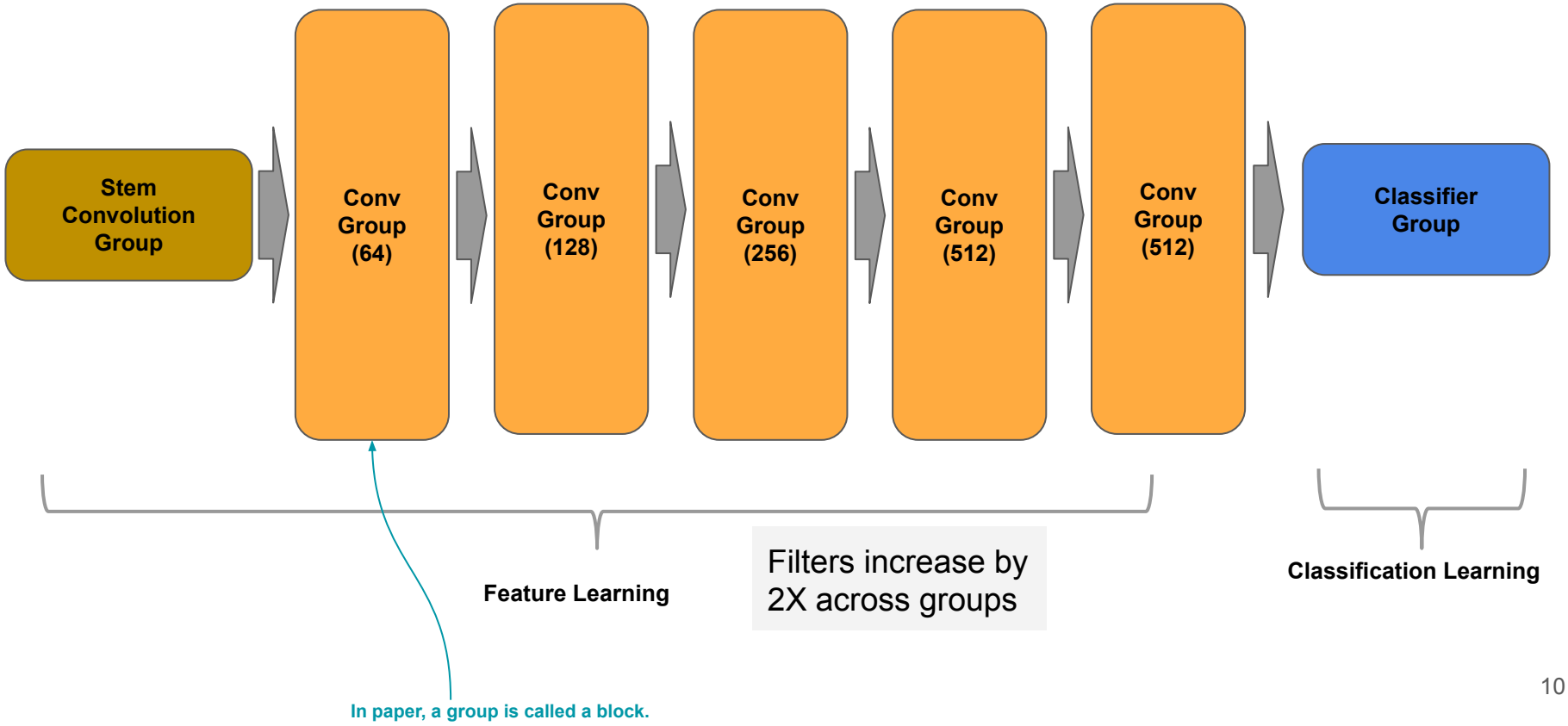


This is data preprocessing part of the Graph (detachable).

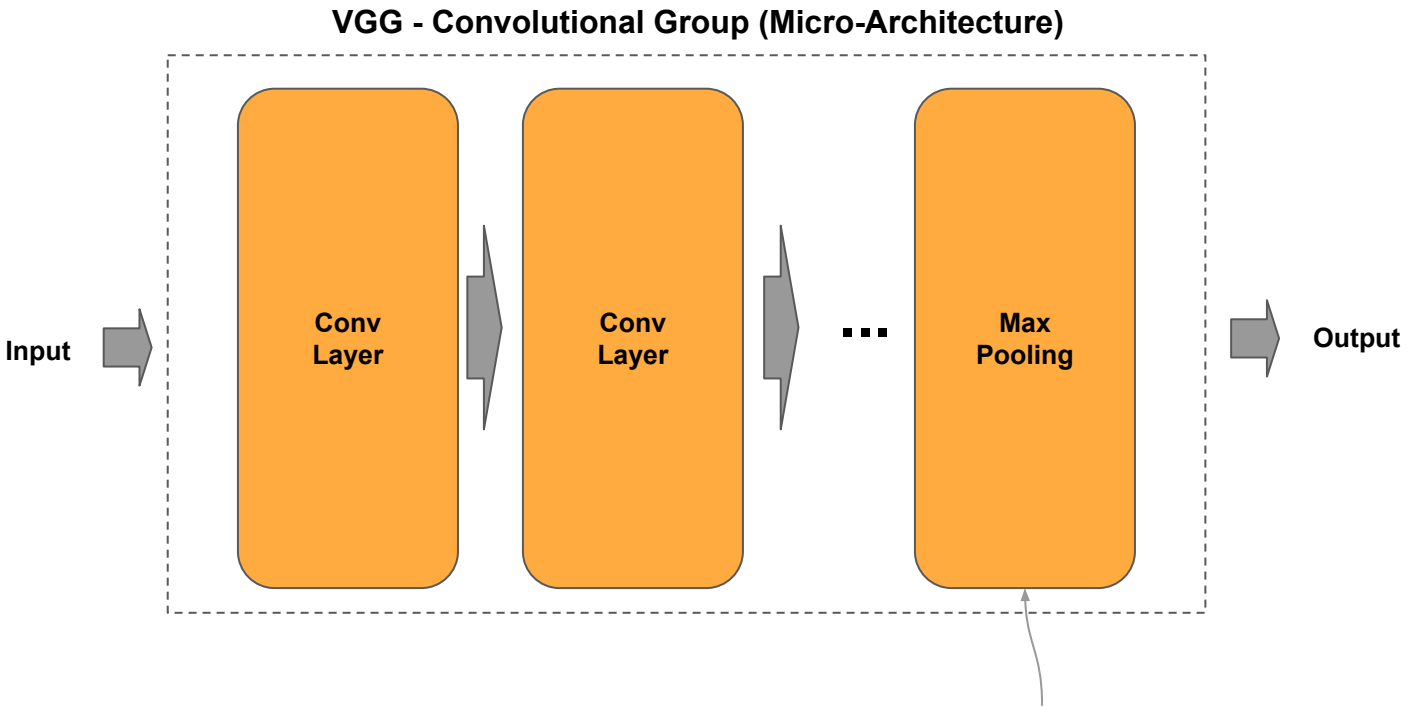
This is the model entry part of the Graph.

Initial convolutional layers for extracting coarse features, followed by pooling the coarse feature maps.

VGG


Stem Convolution Group
Conv Group (64)
Conv Group (128)
Conv Group (256)
Conv Group (512)
Conv Group (512)
Classifier Group
Feature Learning
Classification Learning
Filters increase by 2X across groups
In paper, a group is called a block.

VGG


VGG - Convolutional Group (Micro-Architecture)
Input
Conv Layer
Conv Layer
...
Max Pooling
Output
Delay max pooling to end of convolutional group

VGG

```
def group(inputs, **metaparameters):

    # Block of Layers
    n_filters = metaparameters['n_filters']
    for layer_params in metaparameters['n_layers'']:
        inputs = Conv2D(n_filters, (3, 3), strides=(1, 1),
                        padding='same', activation='relu')(inputs)

    # Max Pooling (downsampling) at end of group
    inputs = MaxPooling2D((2, 2), strides=(2, 2))(inputs)
    return inputs
```

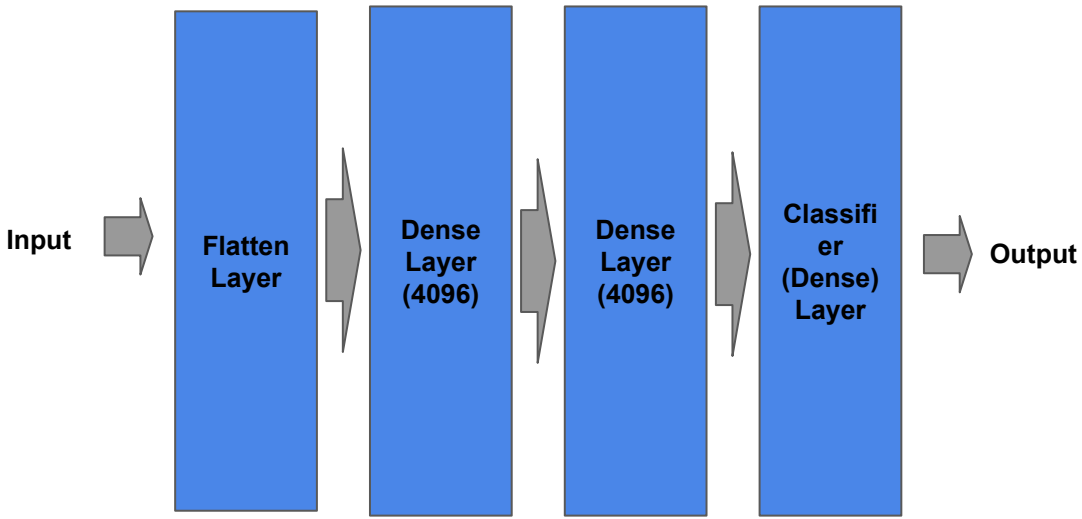
padding='same' preserves size of feature maps:
(H, W)in = (H, W)out

strides=2 reduces height, width by ½:
(H, W)in = (½ H, ½ W)out

VGG

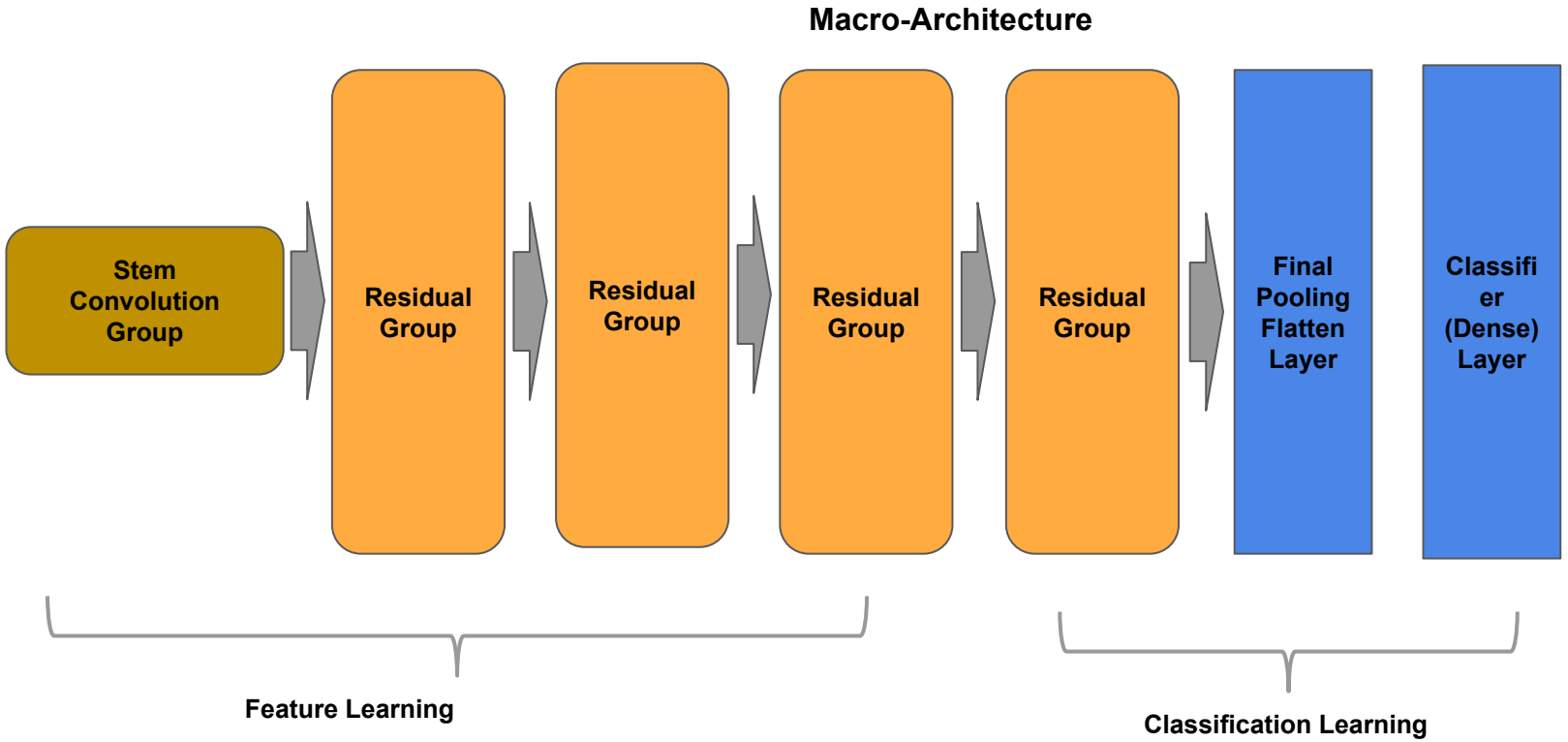
Flattening into 1D vector (lower dimensional embedding) also referred to as the Bottleneck Layer

VGG Classifier Group (Micro-Architecture)



Two very large (4096) dense layers to learn classification from features (1D embedding after Flatten layer)

ResNet

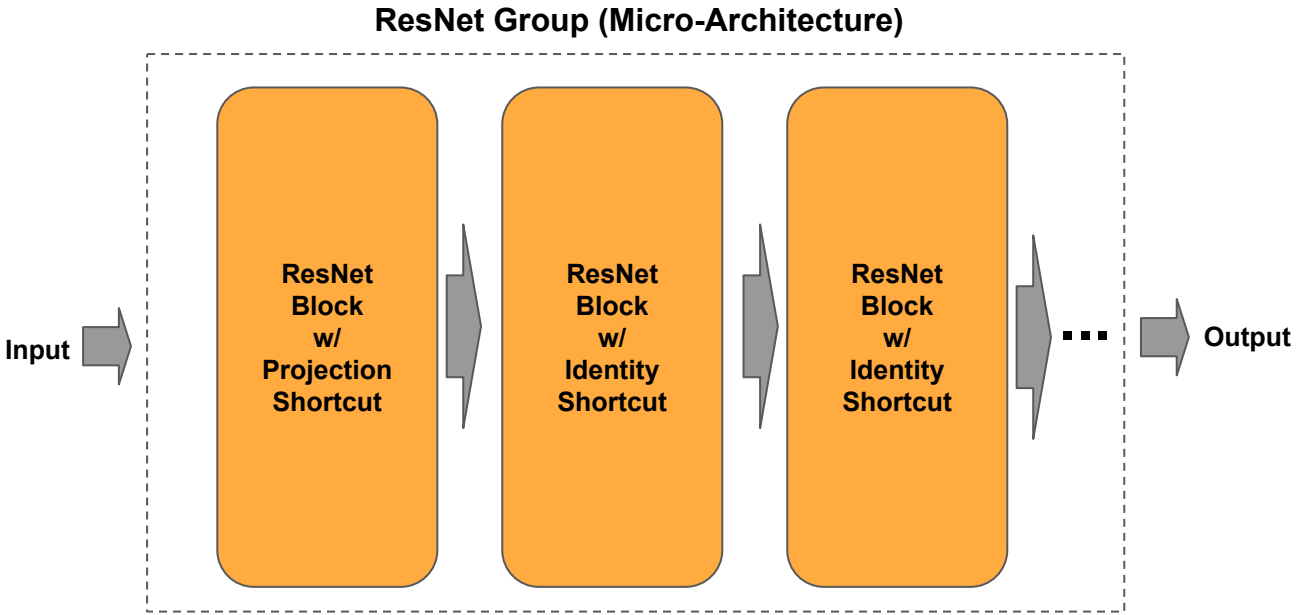


Hidden layers dropped in classifier.

Classification partially moved into top convolutional group

ResNet

Projection shortcut doubles the number of filters.



First block uses linear projection for the residual link to expand the number of feature maps (dimensionality expansion) to match the number of filters for the corresponding group.

ResNet

```
def group(inputs, strides=(2, 2), **metaparameters):

    n_filters = metaparameters['n_filters']
    n_blocks = metaparameters['n_blocks']

    # Linear Projection Block
    inputs = projection_block(inputs, n_filters, strides=strides)

     # Identity Blocks
     for _ in range(n_blocks-1):
         Inputs = identity_block(inputs, n_filters)

     return inputs
```

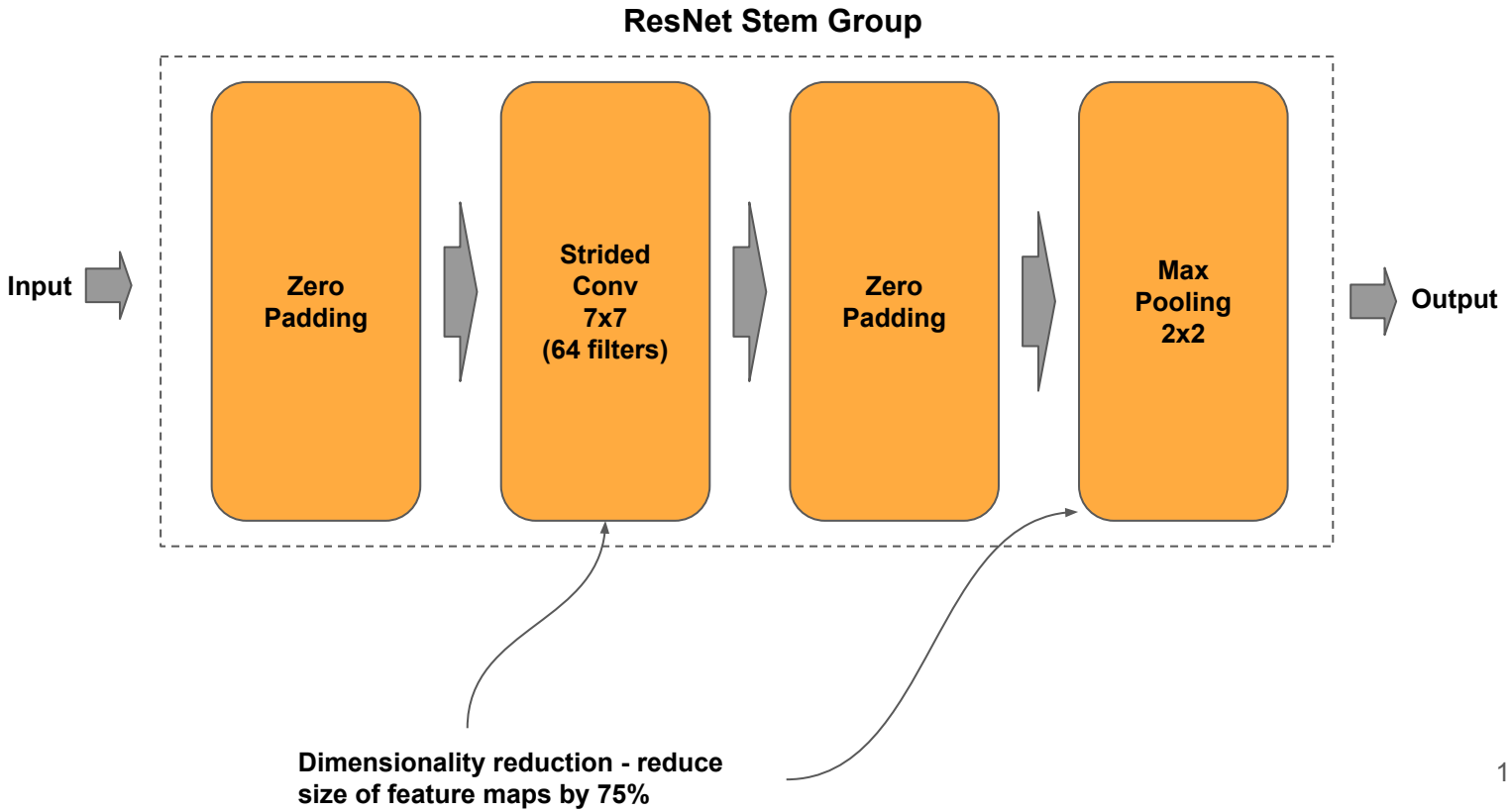
First group does not do feature pooling in projection block --while subsequent groups do feature pooling.

Remaining blocks use identity link (no projection).

ResNet

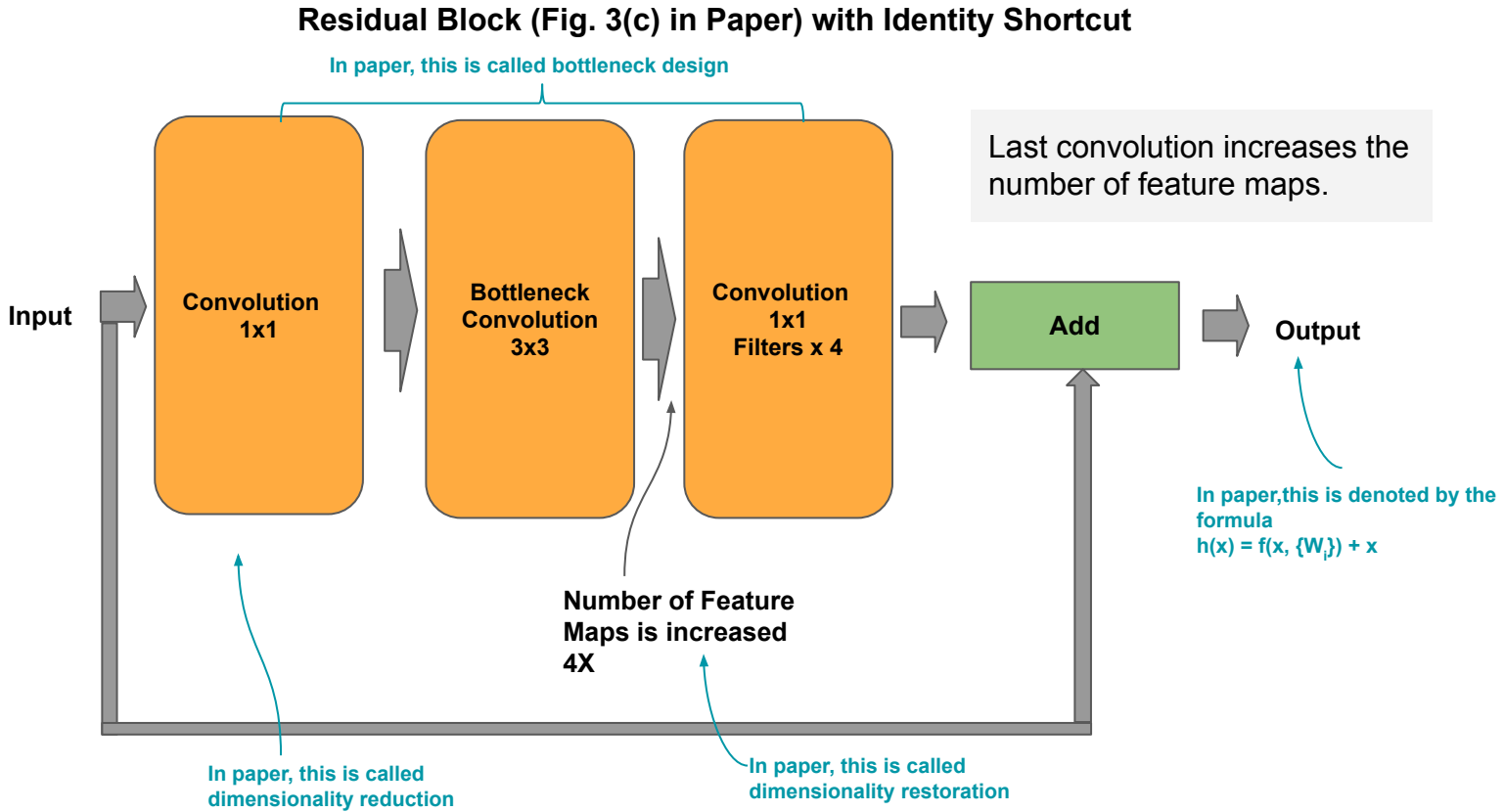
Introduced using a coarse filter size (7x7) vs. VGG (3x3).

Added dimensionality reduction with strided convolution and max pooling.



ResNet

Initial convolution layers reduce the number of features maps from previous block (bottleneck).



ResNet

```
def identity_block(inputs, **metaparameters):
    n_filters = metaparameters['n_filters']

    # Remember the input
    residual = inputs

    # Dimensionality Reduction
    inputs = Conv2D(n_filters, (1, 1), strides=(1, 1), …)(inputs)
    ...

    # Bottleneck Convolution
    inputs = Conv2D(n_filters, (3, 3), strides=(1, 1), …)(inputs)
    …

    # Dimensionality Expansion
    inputs = Conv2D(4 * n_filters, (1, 1), strides=(1, 1), …)(inputs)
    …

    # Add residual block input to output of residual block
    inputs = Add()([residual, inputs])
    return inputs
```

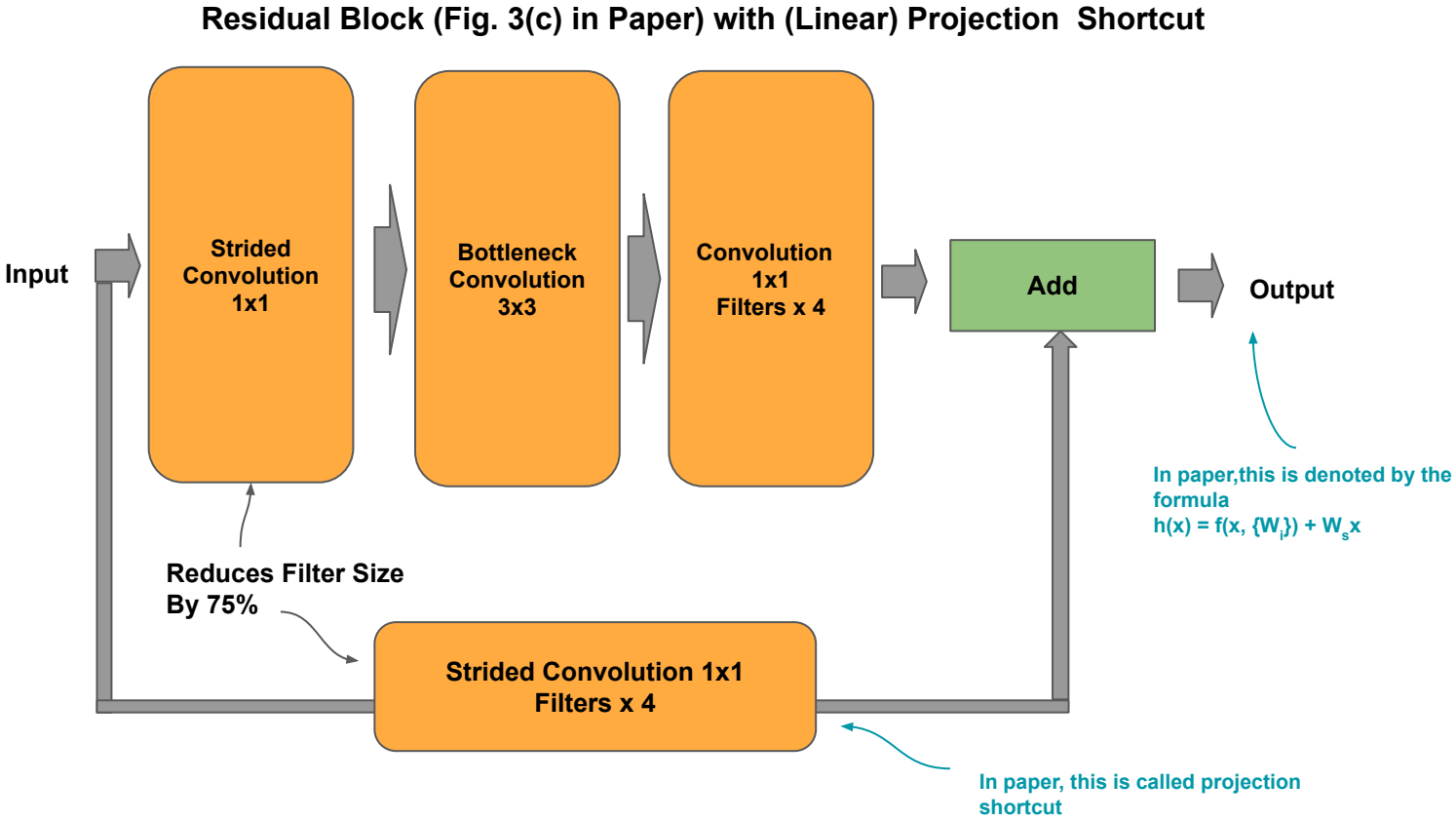
Start by saving a copy of the input (residual).

Do a series of sequential convolutions.

Do a matrix add of the saved input (residual) with outputs of the last convolution.

ResNet

A linear projection convolution is used on the residual in the first block, so the number of feature maps on the identity link match the output for the matrix add operation.



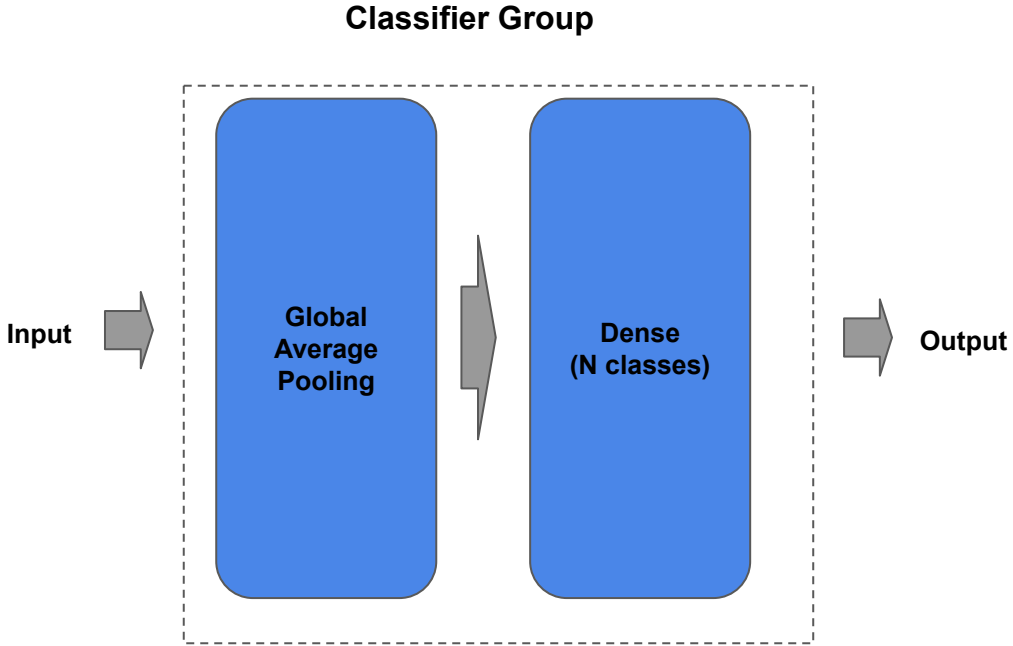
ResNet

```
def projection_block(inputs, strides=(2, 2), **metaparameters):
    n_filters = metaparameters['n_filters']

    # Remember a Linear projection of the inputs
    residual = Conv2D(4 * n_filters, (1, 1), strides=strides, ....)(inputs)

    # Dimensionality Reduction
    inputs = Conv2D(n_filters, (1, 1), strides=(1, 1), …)(inputs)
    ...

    # Bottleneck Convolution
    inputs = Conv2D(n_filters, (3, 3), strides=(1, 1), …)(inputs)
    …

    # Dimensionality Expansion
    inputs = Conv2D(4 * n_filters, (1, 1), strides=(1, 1), …)(inputs)
    …

    # Add residual block input to output of residual block
    inputs = Add()([residual,  inputs])
    return inputs
```

The remembered input (residual) has the number of filters increased 4X on the first block to match the number of filters on the output for the matrix add operation.

ResNet

Flattening of feature maps (bottleneck layer) is replaced by averaging each feature map into a single value and concatenating into 1D vector.



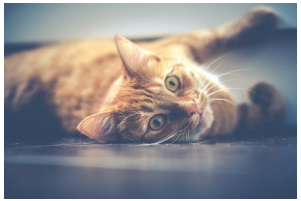
Coarse classification learning overlaps with prior (toplevel) convolutional group.

Final (detail) classification learning is done here.

Batch Normalization

Pixels values are normalized, whereby the distance between pixels is proportional to their frequency of occurrence -- which speeds up learning.

Co-Variance Shift - Vanishing Gradient

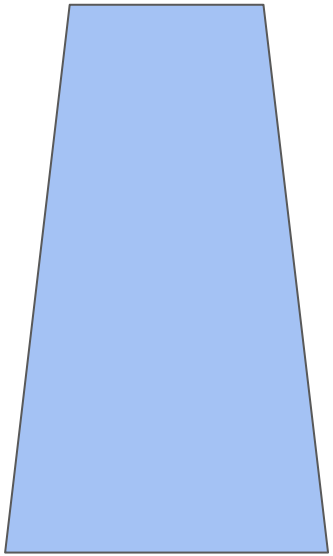


Conv Layer

Conv Layer

Conv Layer

Pixel value spread



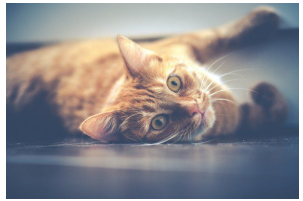
Variance in the pixel values spreads per layer (co-variance).

At some point, the variance is too great for the model to learn - which limited the depth of layers (vanishing gradient).

Batch Normalization

Solution - Renormalize after each convolution

Re-normalize pixel values after each convolution.

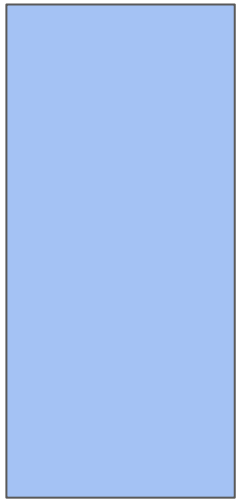


Conv Layer

Conv Layer

Conv Layer

Pixel value spread



Variance in the pixel values stabilizes.

Can go deeper layers without vanishing gradient.

By-product benefit was able to use higher learning rates and speed up training time.

Batch Normalization

```
# Convolutional layer followed by batch normalization
inputs = Conv2D(n_filters, (1, 1), strides=(1, 1), use_bias=False,
                kernel_initializer='he_normal')(inputs)
inputs = BatchNormalization()(inputs)
inputs = ReLU()(inputs)
```

Batch Normalization (normalize over each batch) added inserted before linear activation unit (demonstrated in ResNet).

Eliminated the need for bias parameters (i.e., use_bias = False).

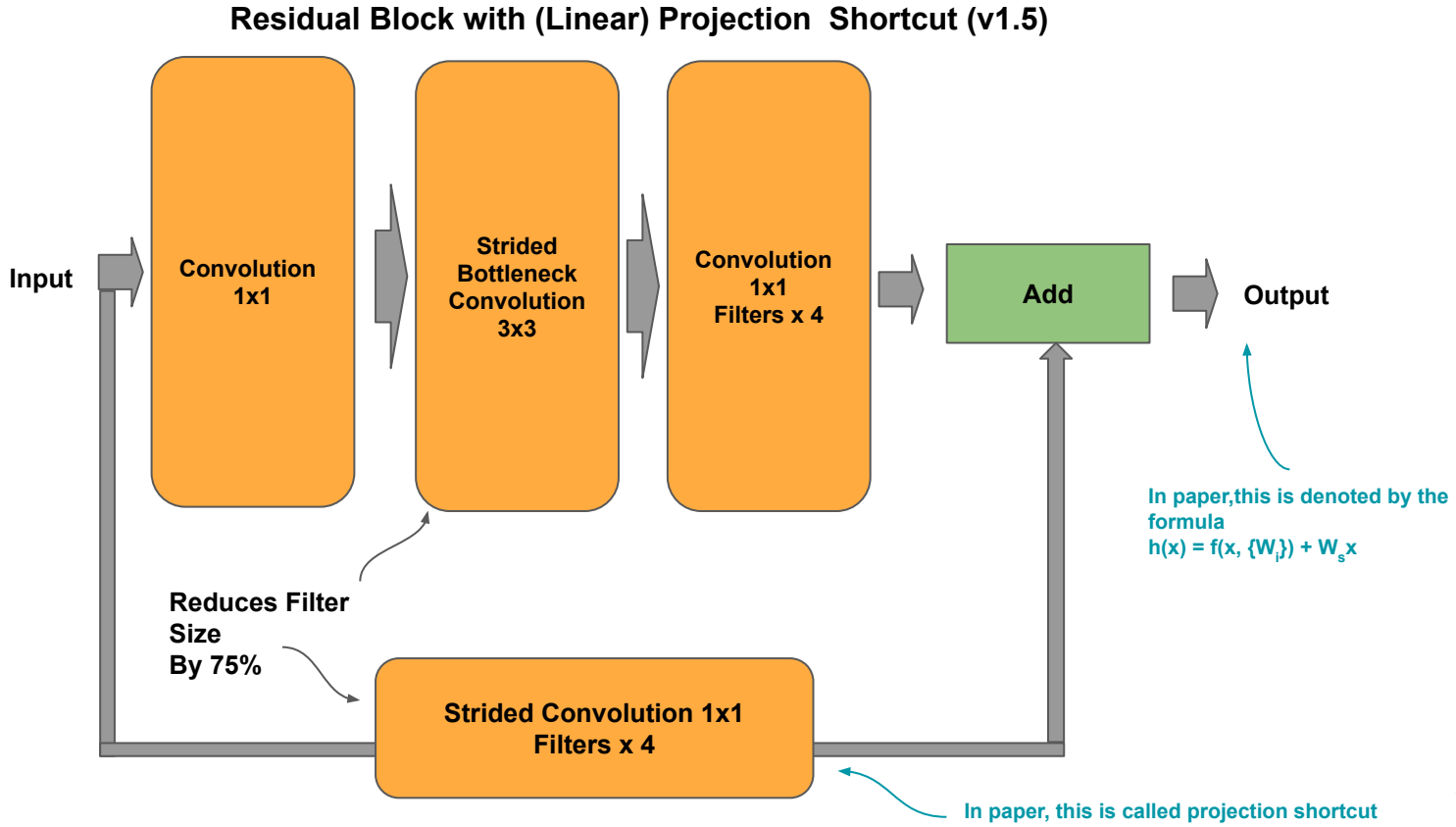
ResNet used random sample from He-Normal distribution for initializing weights (prior was Xavier -- increased likelihood of finding best optima ~ accuracy on holdout data).

ResNet V1.5

The strided convolution in projection block is moved to the 3x3 bottleneck convolution.

Reduces number of multi-ops of the 1x1 / 3x3 pair by 66%. On 4x4 patch, previous was 37 multi-ops, now 13.

Authors claim has representational equivalence with V1 design.



ResNet V2

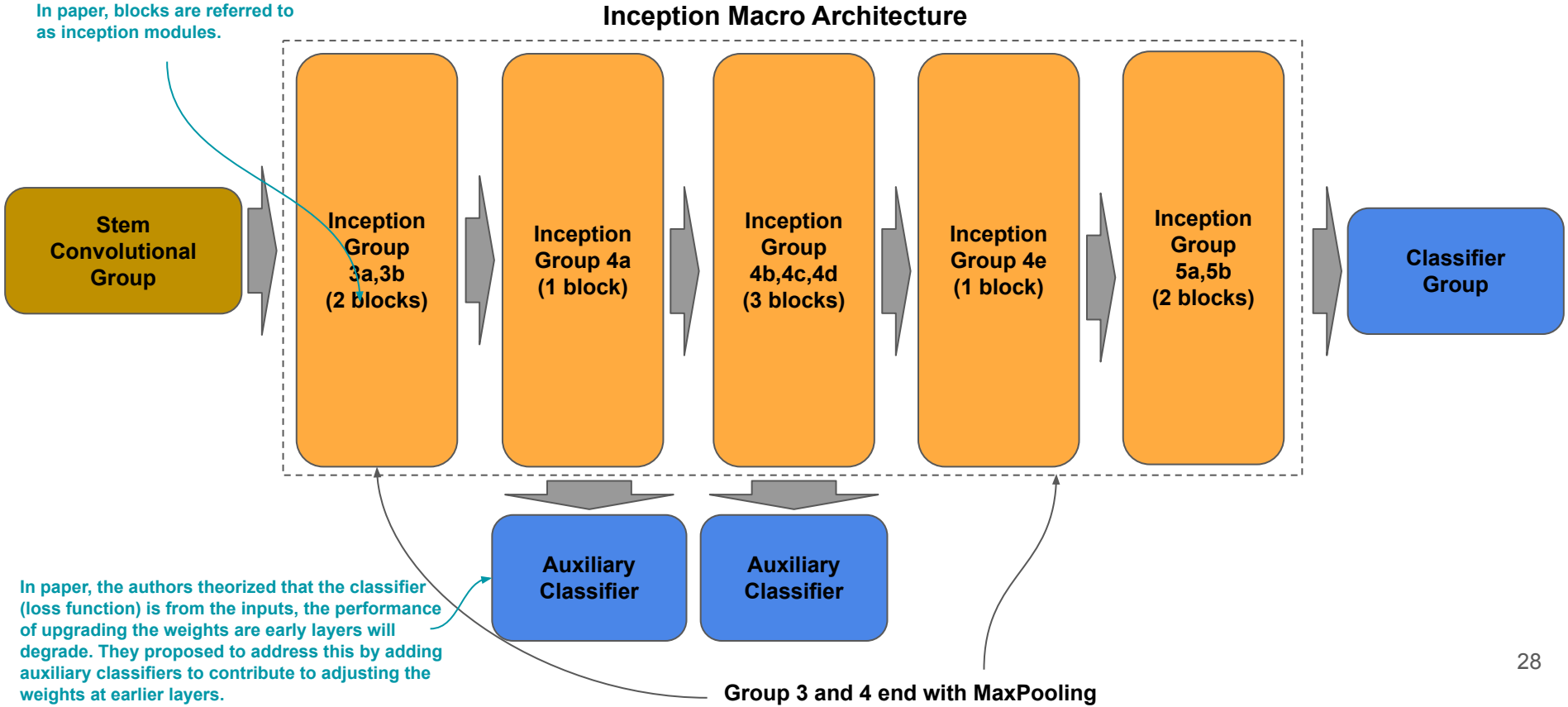
```
# Convolutional layer followed by batch normalization
inputs = BatchNormalization()(inputs)
inputs = ReLU()(inputs)
inputs = Conv2D(n_filters, (1, 1), strides=(1, 1), use_bias=False,
                kernel_initializer='he_normal')(inputs)
```

In V2, the Batch Normalization/ReLU is moved to before the convolution in the identity and projection blocks (but not in the stem). Referred to as BN-RE-Conv pre-activation.

The number of parameters and matmul ops stays the same, while the authors found they got higher accuracies on ImageNet and CIFAR-10 training.

Inception V1 (GoogLeNet)

In paper, blocks are referred to as inception modules.



Inception V1 (GoogLeNet)

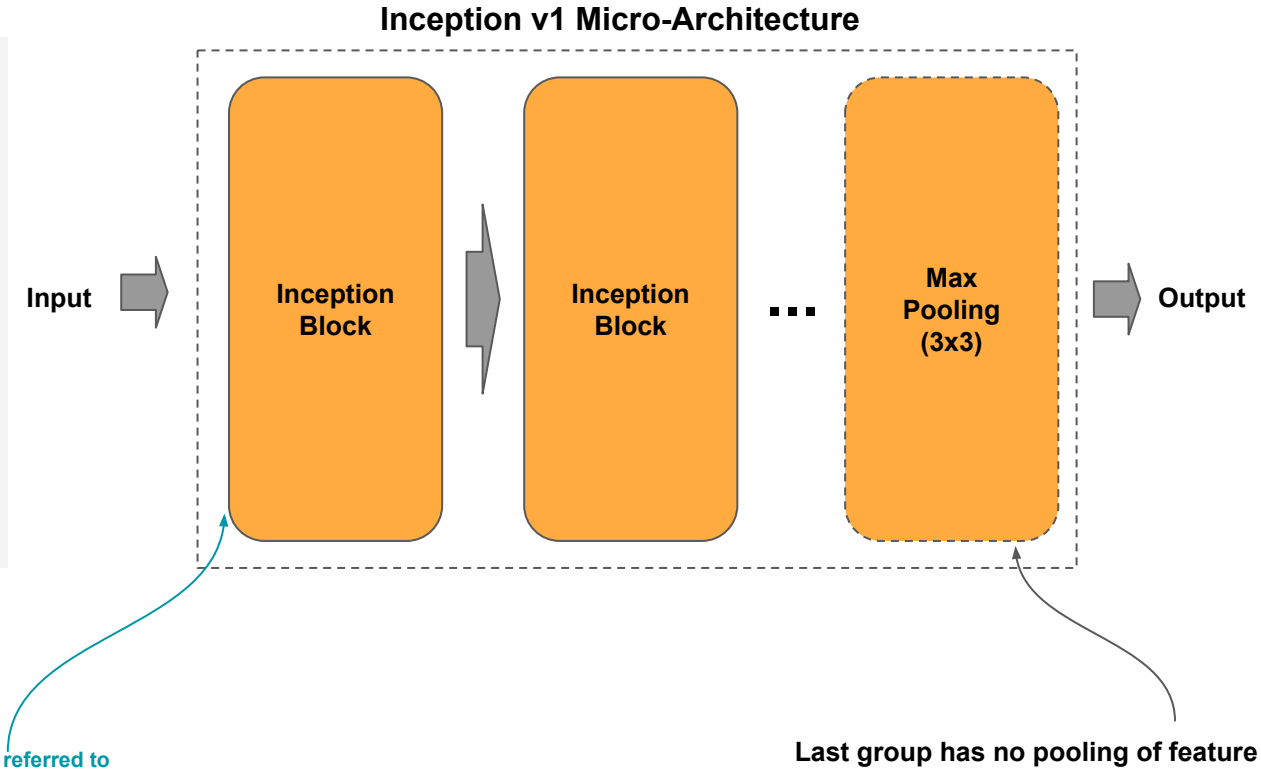
```
def learner(inputs, n_classes, **metaparameters):
    aux = [] # auxiliary outputs

    groups = metaparameters['groups'] # group 3, 4 and 5
    for group_params in groups:
        inputs, _aux = group(inputs, group_params)
        aux += _aux

    return inputs, aux
```

The learner constructs both the sequential convolutional groups (inputs), and the non-sequential auxiliary classifiers (aux).

Inception V1 (GoogLeNet)

The learner consists of three groups, where each group consists of two or more inception blocks (modules) and ends with a max pooling layer for dimensionality reduction between groups.

The total number of filters per group successively increases.



The last group does no have a max pooling layer -- instead dimensionality reduction is done by the bottleneck layer in the classifier.

Inception V1 (GoogLeNet)

```
def group(inputs, pooling=True, **metaparameters):
    aux = [] # auxiliary outputs

    blocks = metaparameters['blocks']
    for block_params in blocks:
        # Add auxiliary classifier after previous block
        if block_params is None:
            aux.append(auxiliary(inputs, n_classes)
        else:
            # Filter sequence for each branch in block
            branch1x1, branch3x3, branch5x5, branchpool = block_params
            inputs = inception_block(inputs, branch1x1, branch3x3, branch5x5,
                                     branchpool)

    # Add max pooling at the end of  the group
    if pooling:
        inputs = ZeroPadding2D((1, 1))(inputs)
        inputs = MaxPooling2D((3, 3), strides=(2, 2))(inputs)

    return inputs, aux
```

Here the metaparameters are a sequential list of blocks and auxiliary classifiers.

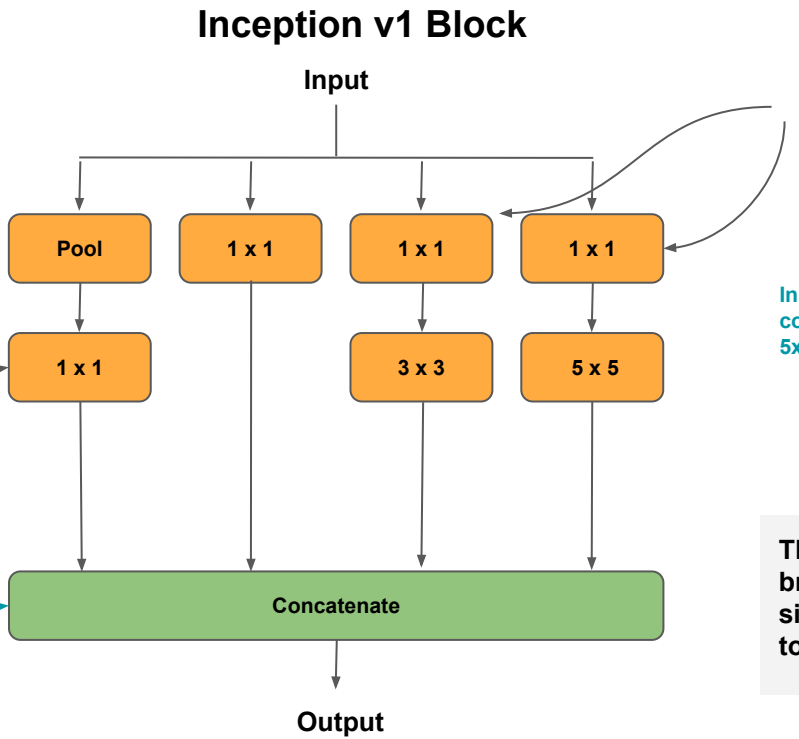
Each inception block (module) uses a wide convolutional with four parallel (branches) convolutions.

The block parameters specify the number of filters per branch.

Inception V1 (GoogLeNet)

The input (feature maps) are passed thru four parallel convolutions (branches) of differing filter sizes.

Authors claimed that the different filter sizes capture different resolution of details.



In paper, this is done to reduce computational complexity on the 3x3 and the more expensive 5x5 filters.

Linear Projection

The output feature maps from each branch are concatenated into a single set of feature maps --referred to as a filter bank.

In paper, this is referred to as a filter bank.

Inception V1 (GoogLeNet)

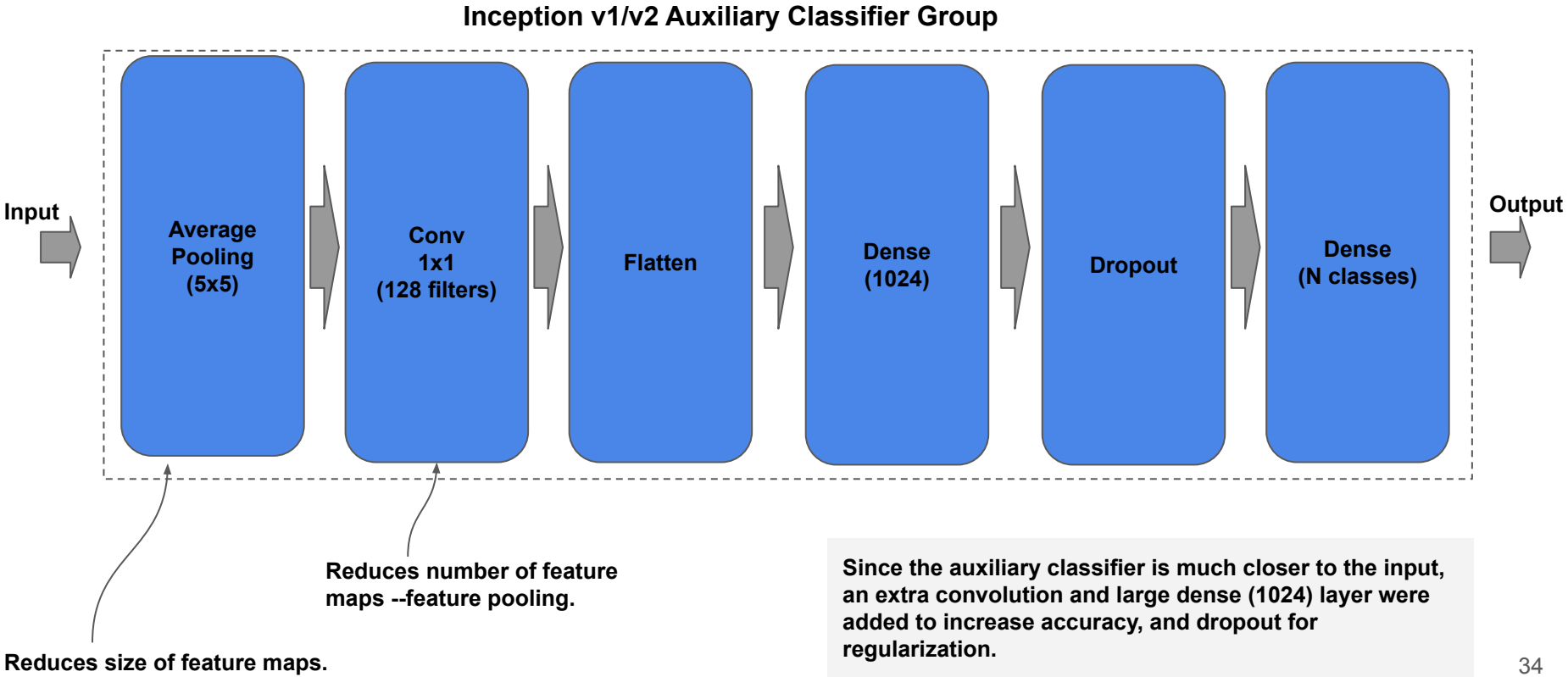
```
def inception_block(inputs, f1x1, f3x3, f5x5, fpool):
    # The branches
    b1x1 = Conv2D(f1x1, (1, 1), strides=1, padding='same', activation='relu')(inputs)

    b3x3 = Conv2D(f3x3[0], (1, 1), strides=1, padding='same', activation='relu')(inputs)
    b3x3 = ZeroPadding2D((1, 1))(b3x3)
    b3x3 = Conv2D(f3x3[1], (3, 3), strides=1, padding='valid', activation='relu')(b3x3)

    b5x5 = Conv2D(f5x5[0], (1, 1), strides=1, padding='same', activation='relu')(inputs)
    b5x5 = ZeroPadding2D((1, 1))(b5x5)
    b5x5 = Conv2D(f3x3[1], (3, 3), strides=1, padding='valid', activation='relu')(ib5x5)

    bpool = MaxPooling2D((3, 3), strides=1)(inputs)
    bpool = Conv2D(fpool, (1, 1), strides=1, padding='same', activation='relu')(bpool)

    # The filter bank
    inputs = Concatenate()([b1x1, b3x3, b5x5, bpool])

    return inputs
```

The input is passed thru four parallel convolutions of different filter sizes.

Zero padding used to preserve the size of the feature maps (i.e., all branches have the same size) for subsequent concat.

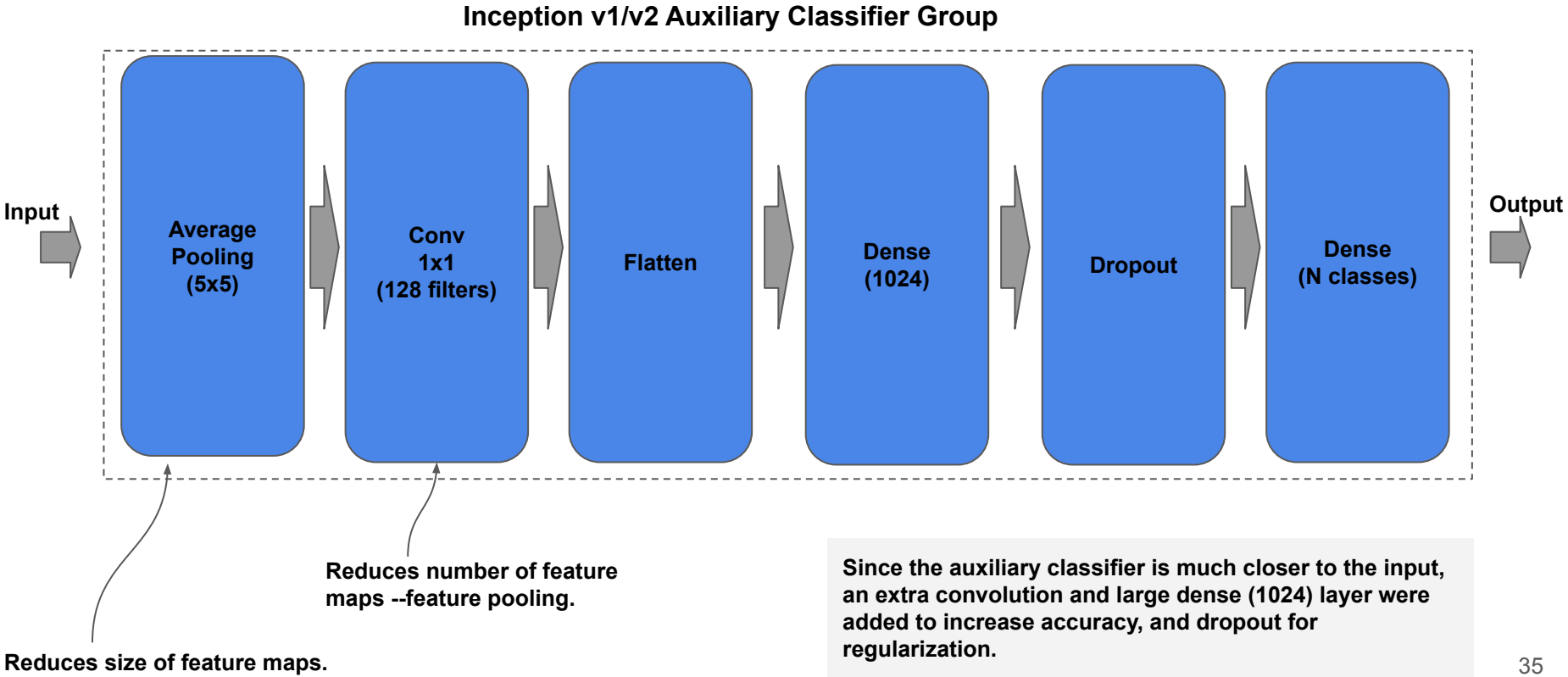
Concatenate the features maps from the branches into a filter bank.

Inception V1 (GoogLeNet)



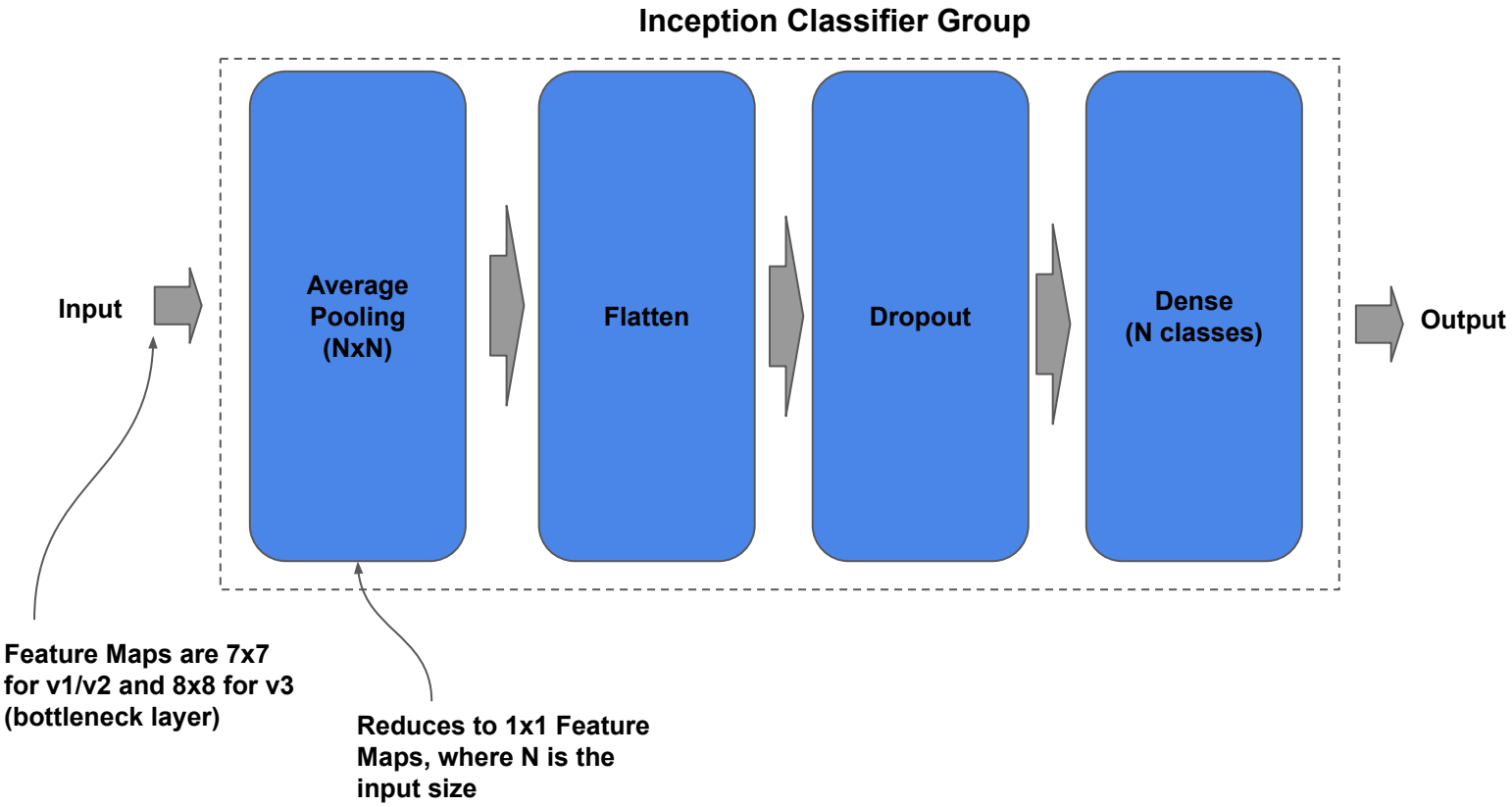
Since the auxiliary classifier is much closer to the input, an extra convolution and large dense (1024) layer were added to increase accuracy, and dropout for regularization.

Inception V1 (GoogLeNet)



Since the auxiliary classifier is much closer to the input, an extra convolution and large dense (1024) layer were added to increase accuracy, and dropout for regularization.

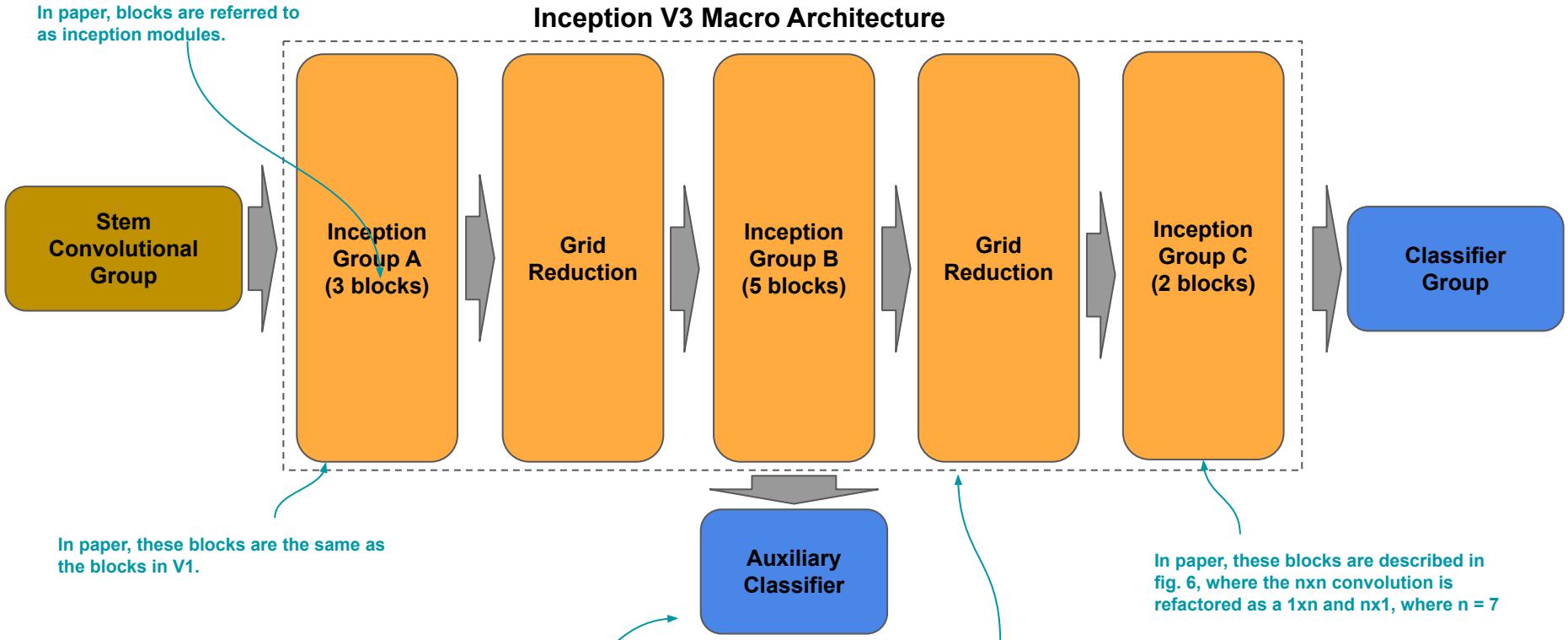
Inception V1 (GoogLeNet)



Classifier reduced to bottleneck layer (pooling / flattening) and final dense layer.

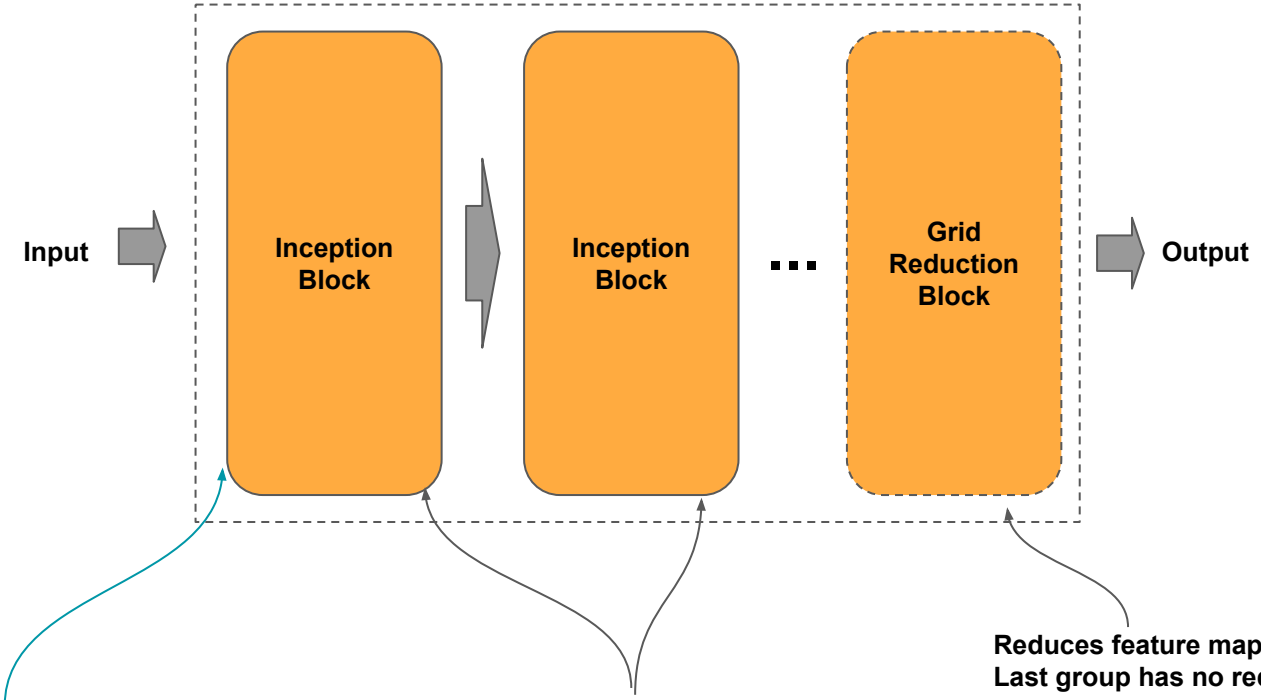
Dropout added for regularization.

Inception V3



Inception V3

V3 has three styles of inception blocks, referred to as 35x35 (group A), 17x17 (group B) and 8x8 (group C).



V3 replaces max pooling at the end of a group with a grid reduction block for feature pooling (dimensionality reduction).

Inception V3

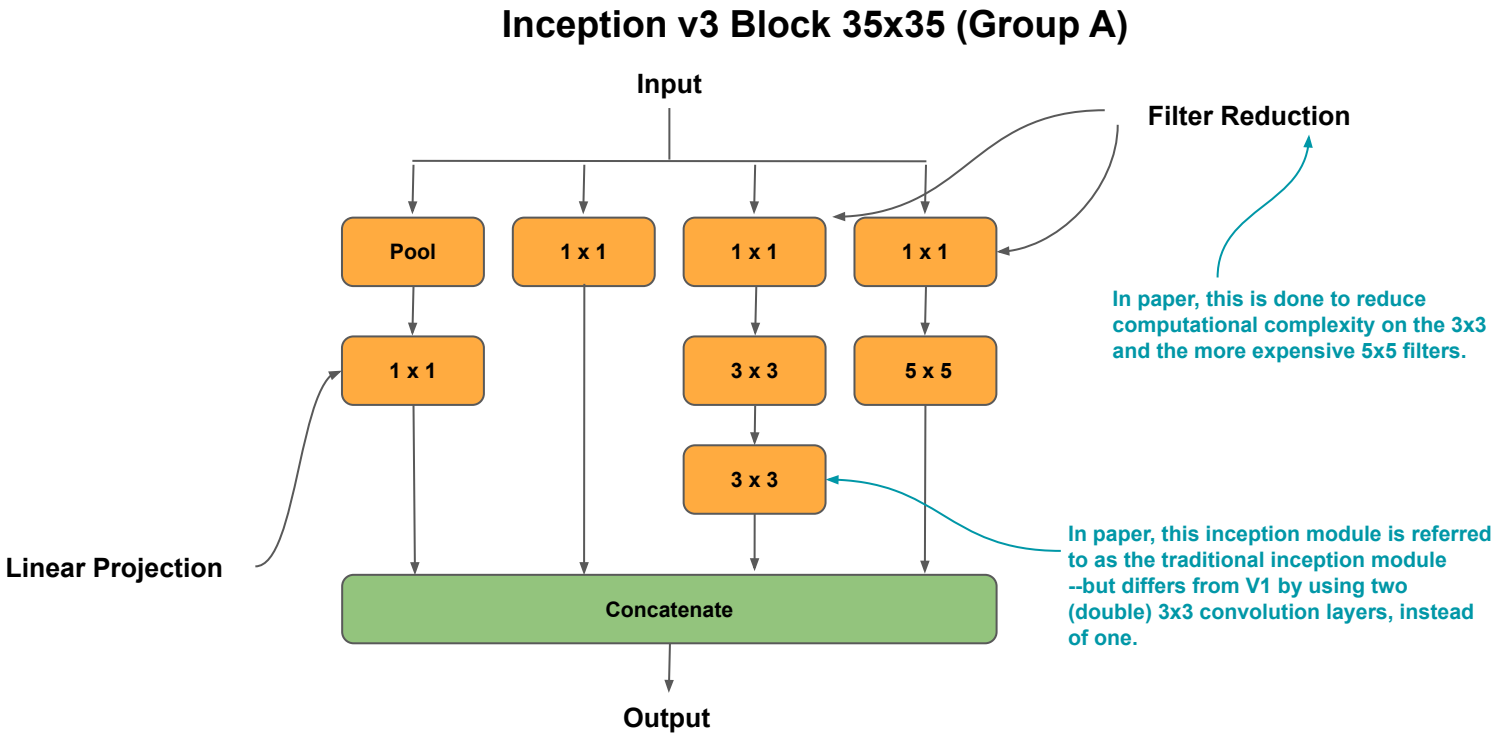
```
def group(inputs, **metaparameters):
    # The style of inception block for this group
    inception_block = metaparameters['inception']
    # The style of grid reduction (or None) for this group
    grid_reduction  = metaparameters['reduction']
    # Include auxiliary classifier (or None) for this group
    auxiliary_classifier = metaparameters['auxiliary']

    for block_params in metaparameters['blocks']:
        # The number of filters per branch
        branch1, branch2, branch3, branch4 = block_params
        inputs = inception_block(inputs, branch1, branch2, branch3, branch4)

    if auxiliary_classifier is not None:
        aux = auxiliary(inputs, n_classes=auxiliary_classifier)

    if grid_reduction is not None:
        Inputs = grid_reduction(inputs)

    return inputs, aux
```

The metaparameters consist of the style of inception block and grid reduction, and whether to include the auxiliary classifier.

All three styles of inception block use four branches.

Grid reduction done at the end of each group, except the last group.

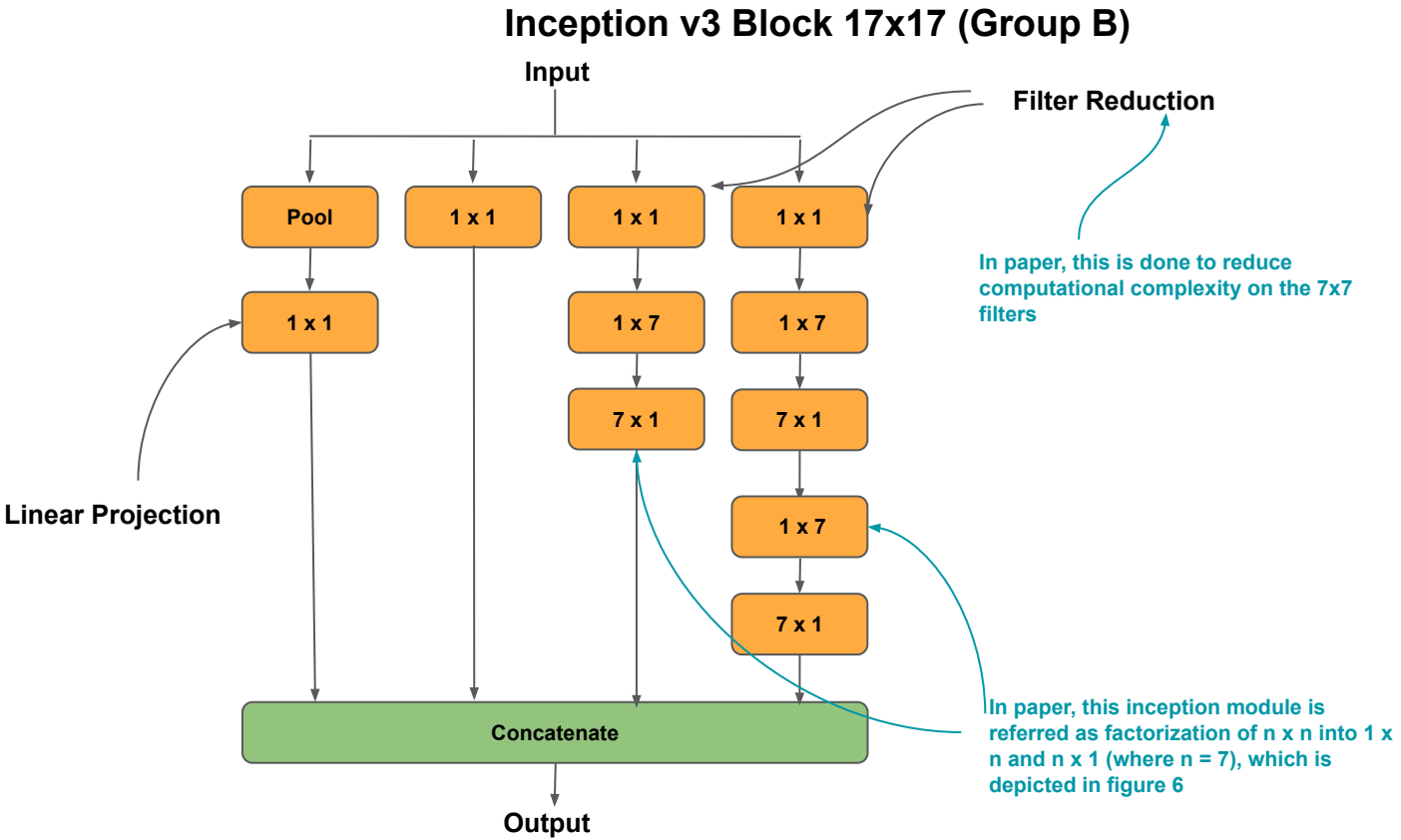
Inception V3



The single 3x3 in inception V1 is replaced by a double 3x3.

While adding computational complexity, the authors state that it increases representational power at small increase in computation.

Inception V3

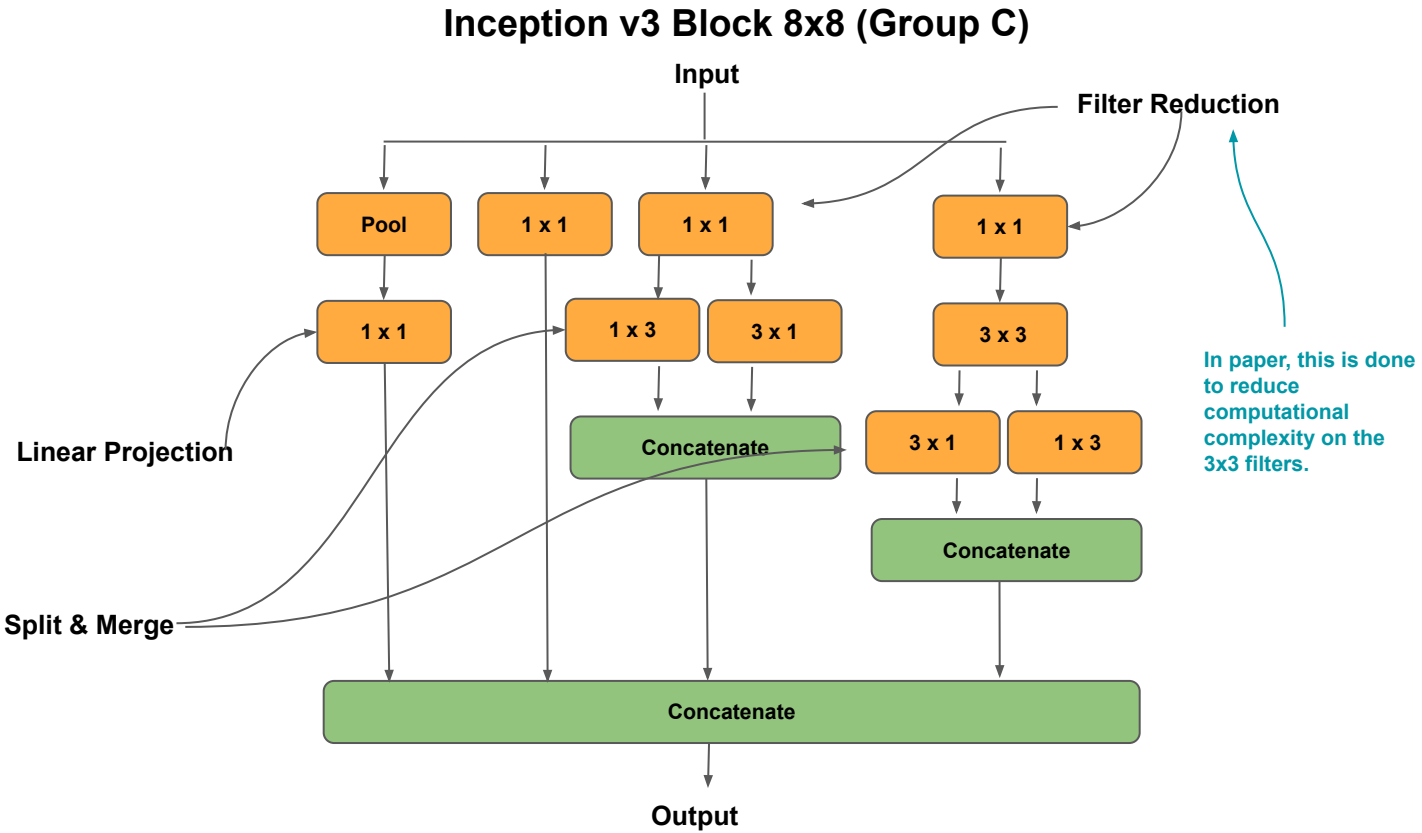


Factorizes the 5x5 convolution into a spatially separable convolution of 1x7, 7x1. Lowers computational complexity from 25 matmul ops to 14 per stride.

Factorizes the double 3x3 convolution into two 1x7, 7x1. Increase computational complexity from 18 matmul ops to 14.

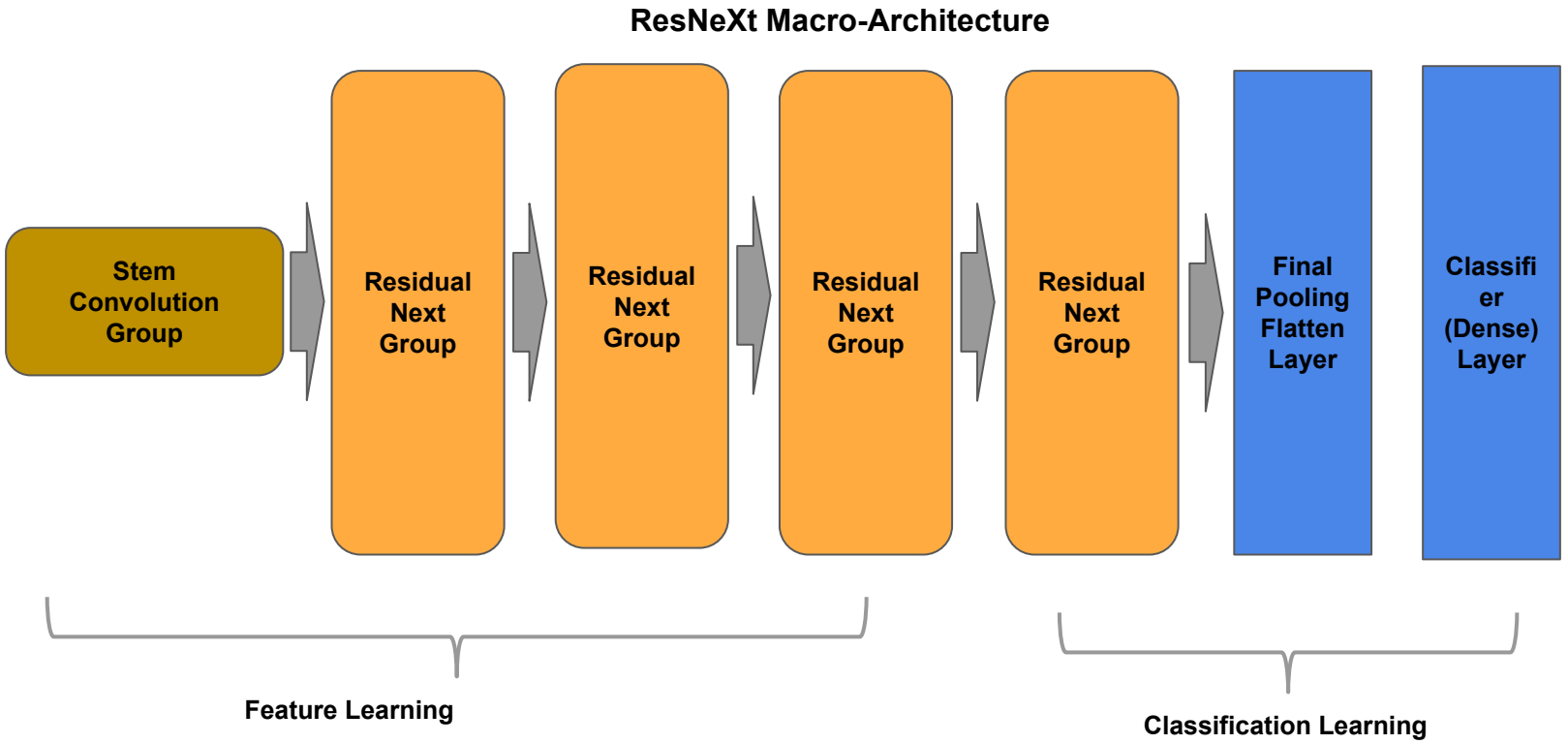
Authors state it increases representational power.

Inception V3



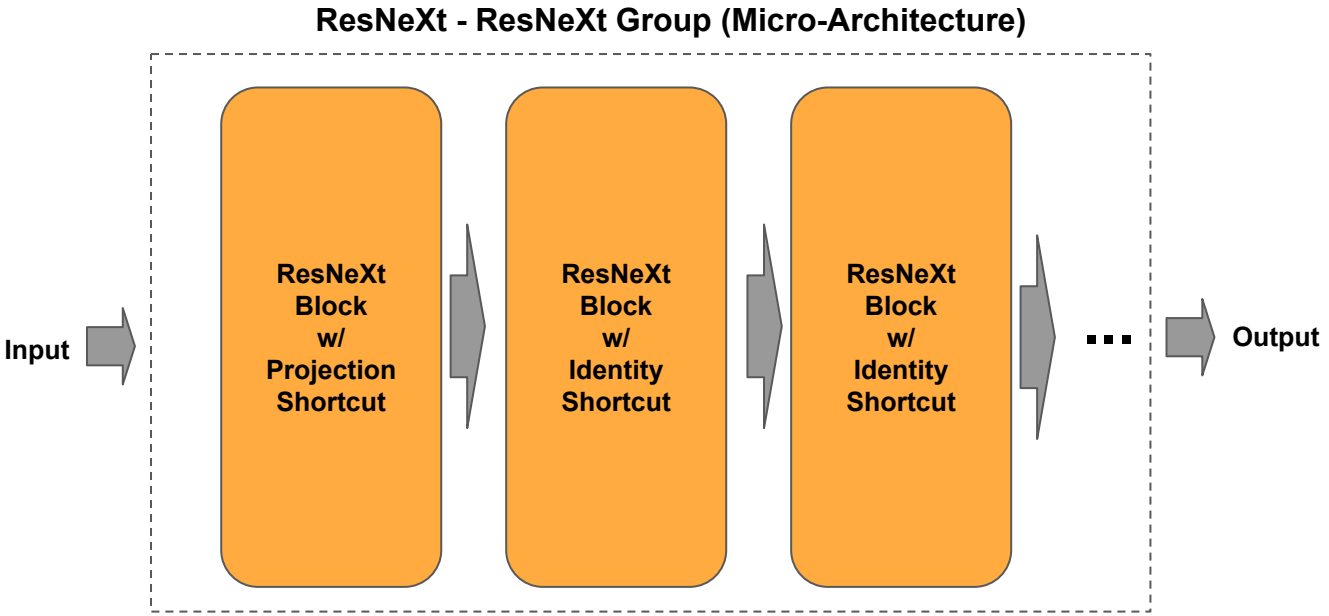
Factorizes the 3x3 convolution into parallel spatially separable convolution 1x3, 3x1.

ResNeXt


ResNeXt Macro-Architecture
Stem Convolution Group
Residual Next Group
Residual Next Group
Residual Next Group
Residual Next Group
Final Pooling Flatten Layer
Classifier (Dense) Layer
Feature Learning
Classification Learning

ResNeXt

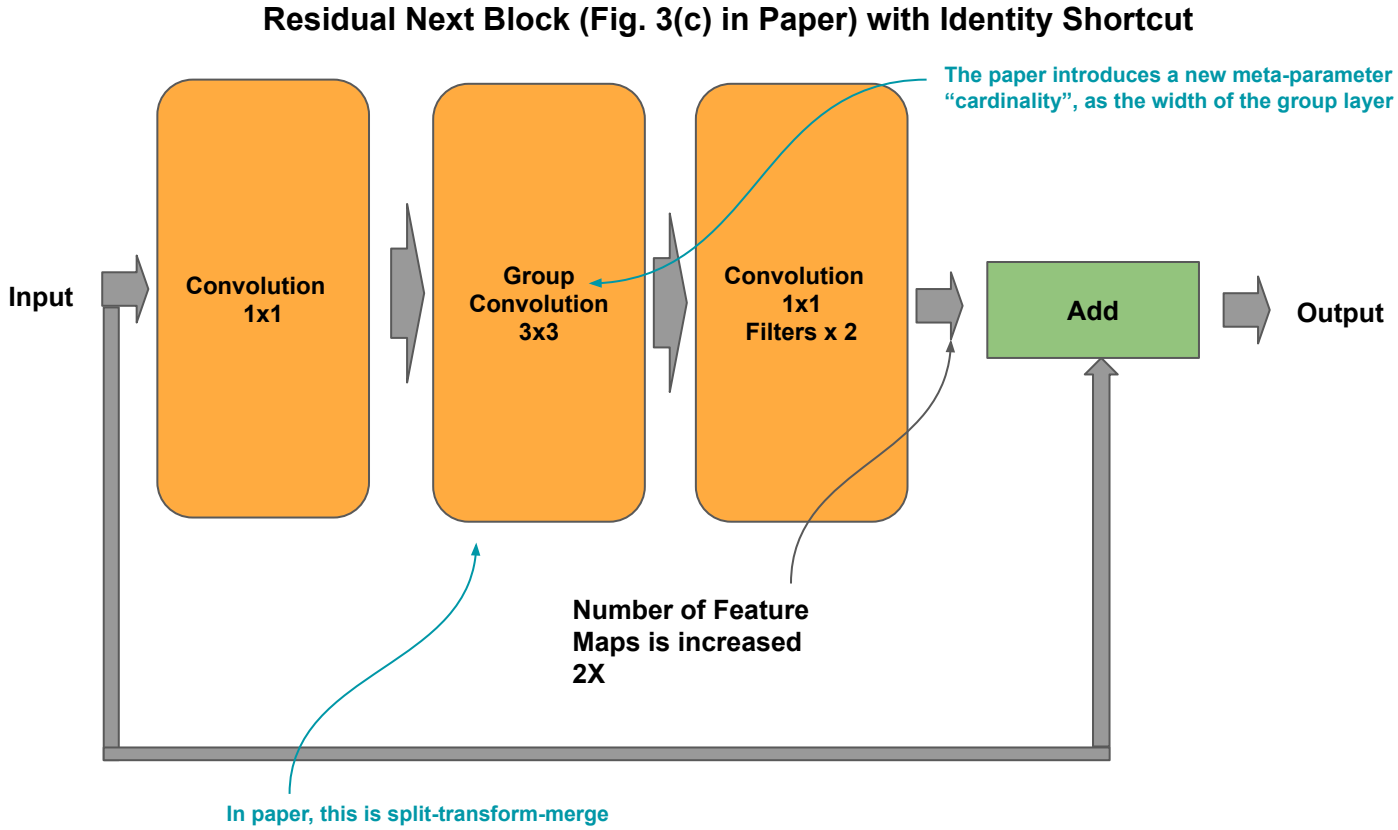
Projection shortcut doubles the number of filters coming in.



Uses a Residual Next block in place of Residual block (ResNet).

Output filters is two times the number of input filters.

ResNeXt



The 3x3 convolution in a Residual block is replaced by a 3x3 group convolution.

The feature maps are split into N segments (cardinality), where each goes through a separate convolution.

ResNeXt

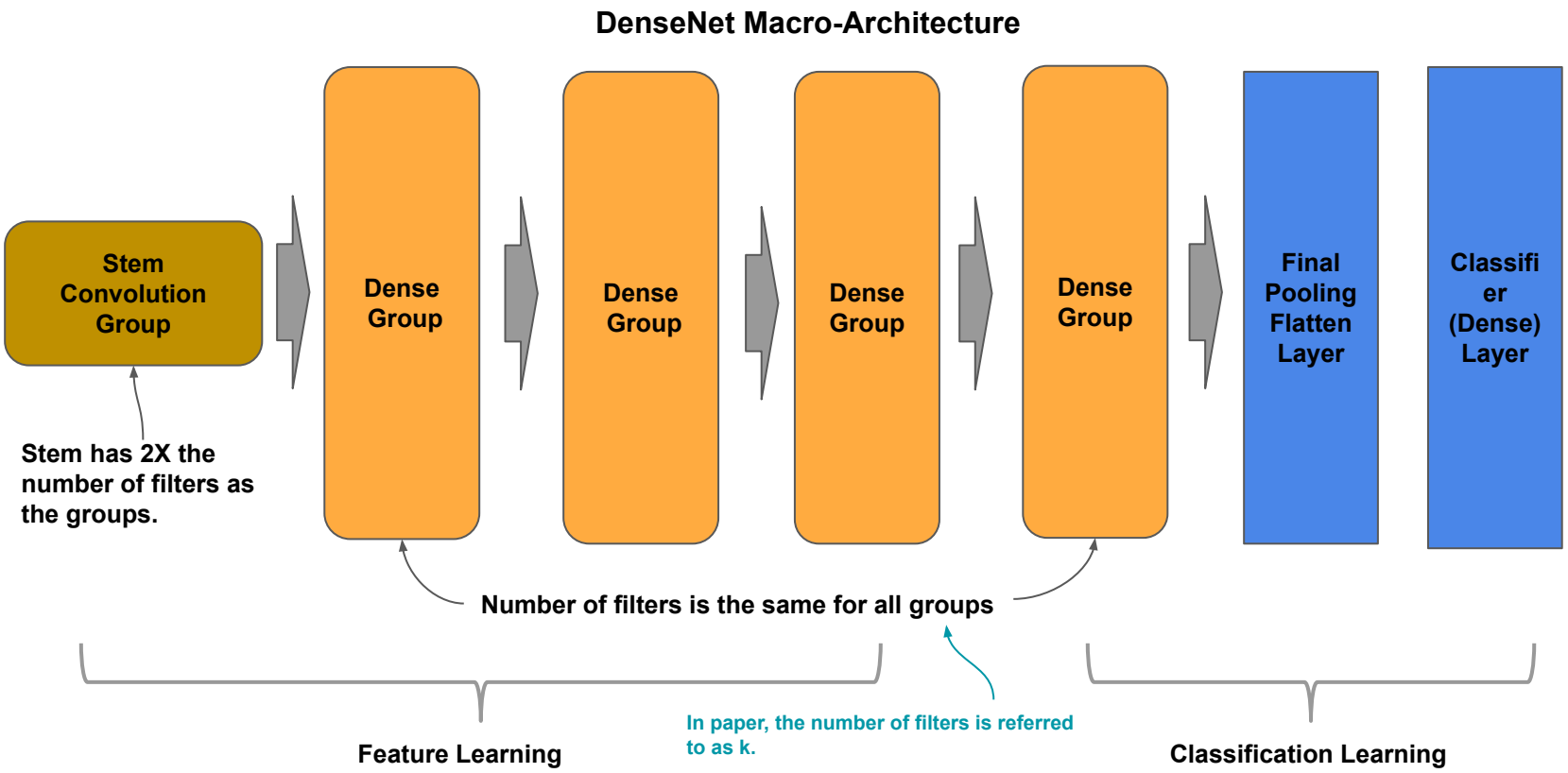
```
def identity_block(inputs, filters_in, filters_out, cardinality=32):
    # removed for brevity ...
    # calculate the number of filters per group
    filters_group = filters_in // cardinality

    # wide layer (split-transform)
    groups = []
    for i in range(cardinality):
        # calculate start/end of partition for the group
        start = i * filters_group
        end  = start + filters_group
        group = Lambda(lambda z: z[:, :, :, start : end])(inputs)
        groups.append(Conv2D(filters_card, (3, 3), strides=(1, 1), padding='same',
                             use_bias=False)(group)

    # merge
    inputs = Concatenate()(groups)
    inputs = BatchNormalization()(inputs)
    # removed for brevity …

    return inputs
```

We use a TF.Keras Lamda layer to perform the splitting of the feature maps during training/inference in the graph.

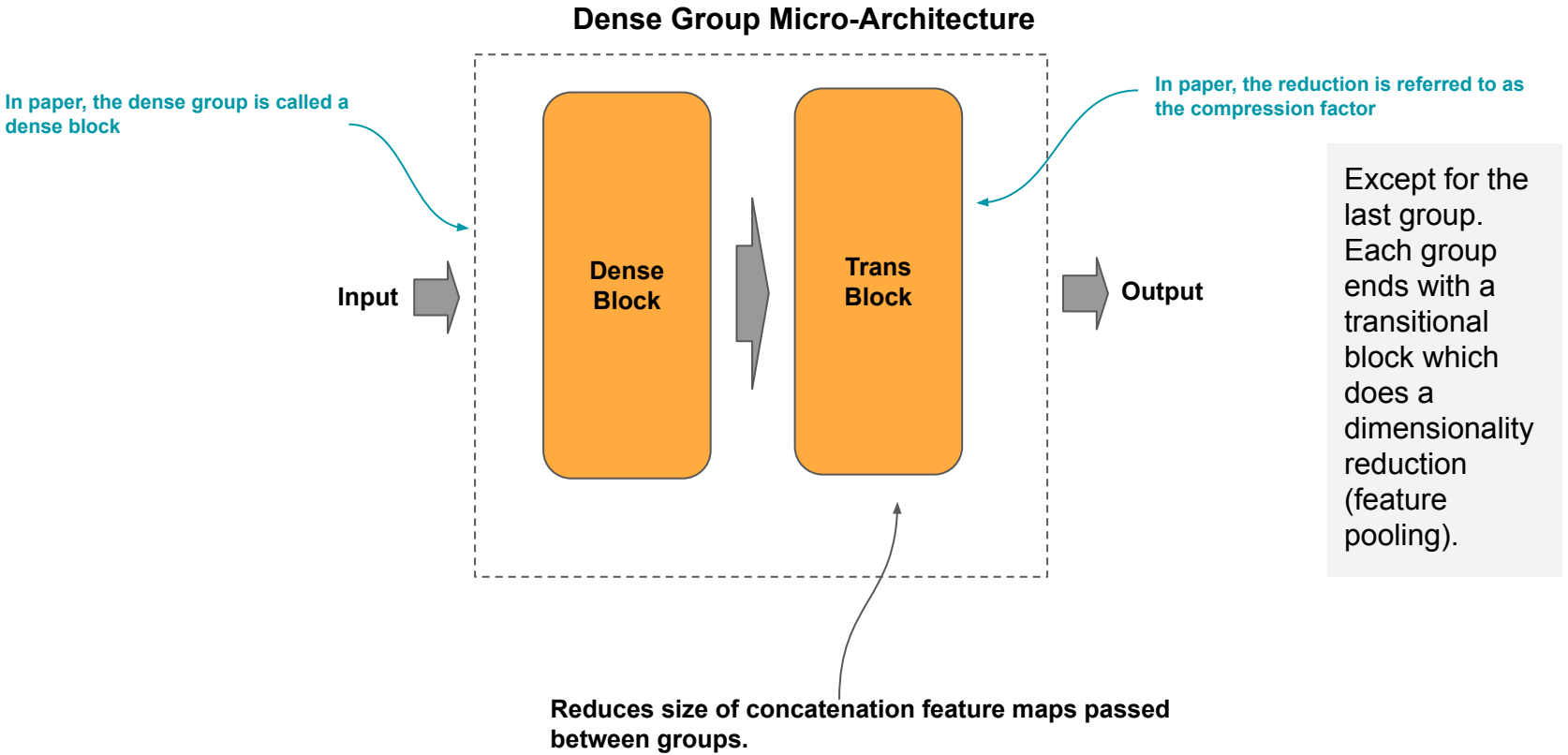
The convolutional outputs of each group are then concatenated together.

The convolutions can be processed in parallel on a GPU (CUDA).

DenseNet


DenseNet Macro-Architecture
Stem Convolution Group
Dense Group
Dense Group
Dense Group
Dense Group
Final Pooling Flatten Layer
Classifier (Dense) Layer
Stem has 2X the number of filters as the groups.
Number of filters is the same for all groups
In paper, the number of filters is referred to as k.
Feature Learning
Classification Learning

DenseNet


Dense Group Micro-Architecture
In paper, the dense group is called a dense block
In paper, the reduction is referred to as the compression factor
Input
Dense Block
Trans Block
Output
Except for the last group. Each group ends with a transitional block which does a dimensionality reduction (feature pooling).
Reduces size of concatenation feature maps passed between groups.

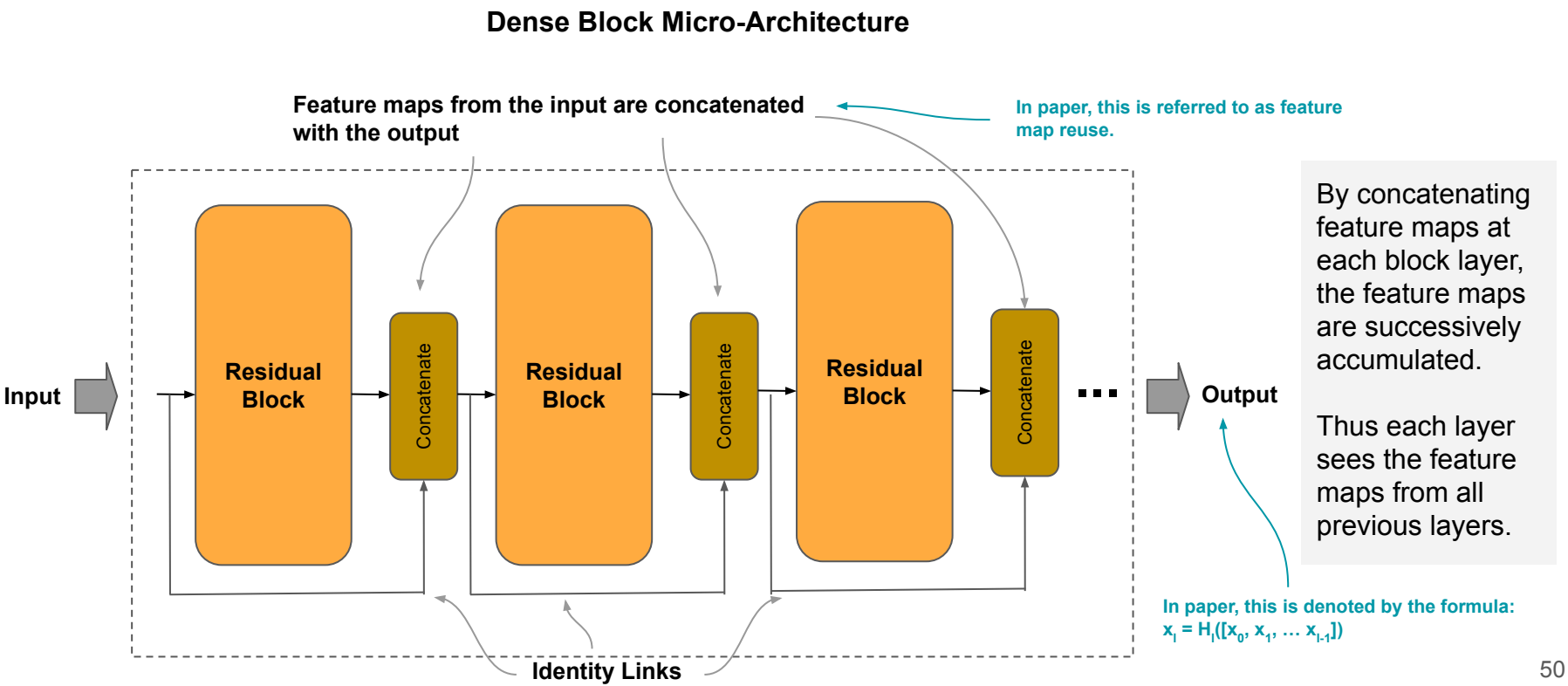
DenseNet

```
def group(inputs, **metaparameters):
      # The amount of reduction (compression factor) or None
      reduction  = metaparameters['reduction']
      # Parameters for residual sub-blocks with dense block
      blocks = metaparameters['blocks']:

      inputs = dense_block(inputs, blocks)

      if reduction is not None:
            inputs = trans_block(inputs, reduction)

      return inputs
```

The metaparameters consist of reduction (compression factor), and the number of filters per block.

Reduction (compression) done at the end of each group, except the last group.

DenseNet



By concatenating feature maps at each block layer, the feature maps are successively accumulated.

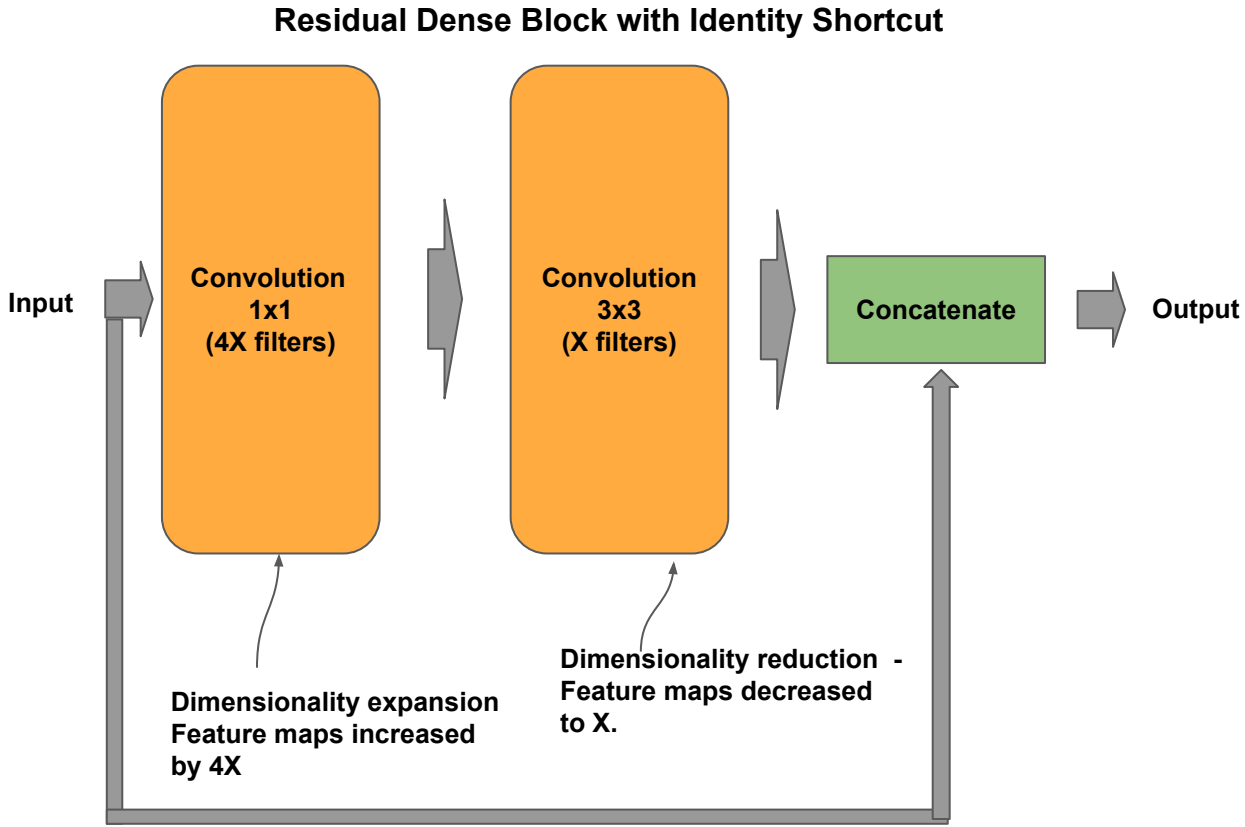
Thus each layer sees the feature maps from all previous layers.

DenseNet

```
def dense_block(inputs, blocks):
    # Parameters for residual sub-blocks with dense block
    blocks = metaparameters['blocks']:

    for block_params in blocks:
        # block_params is the number of filters
        n_filters = block_params
        inputs = residual_dense_block(inputs, n_filters)

    return inputs
```

The concatenation operation occurs within the residual dense block, as the final step.

DenseNet



The feature maps from the input (identity link) are concatenated with the output feature maps.

The accumulated feature maps will then be the input to the next block.

DenseNet

```
def residual_dense_block(inputs, n_filters):
    # remember the input
    residual = inputs

    # dimensionality expansion
    inputs = BatchNormalization()(inputs)
    inputs = ReLU()(inputs)
    inputs = Conv2D(4 * n_filters, (1, 1), strides=(1, 1), use_bias=False)(inputs)

    # bottleneck convolution
    inputs = BatchNormalization()(inputs)
    inputs = ReLU()(inputs)
    Inputs = Conv2D(n_filters, (3, 3), striddes=(1, 1), padding='same',
                        use_bias=False)(inputs)

    # Concatenate residual block input to output of residual block
    inputs = Concatenate()([residual, inputs])
    return inputs
```

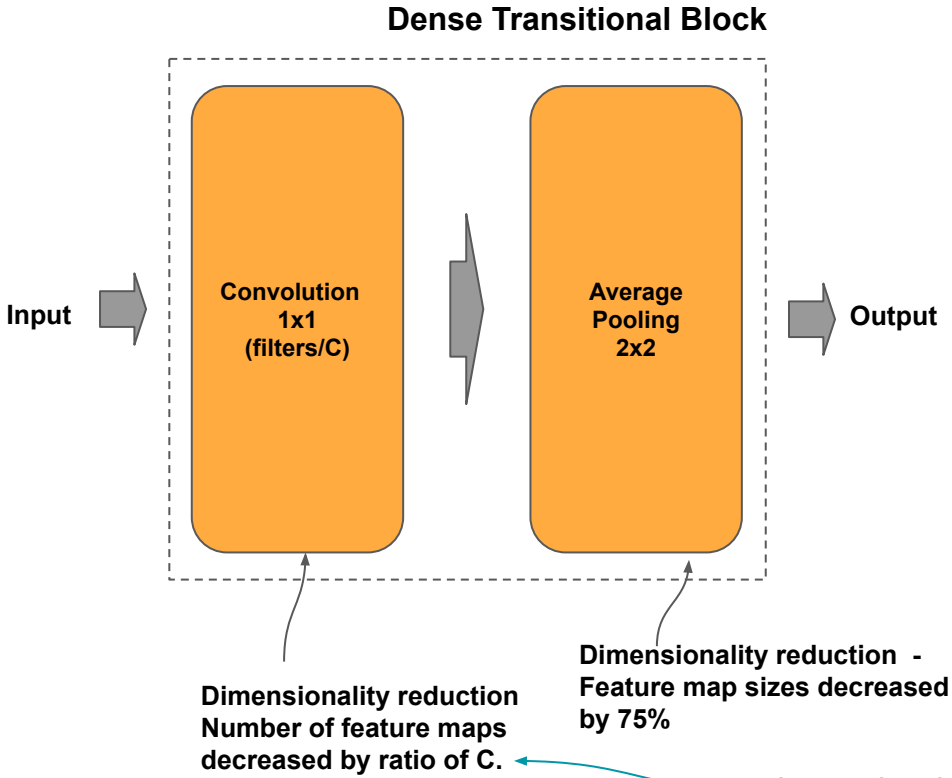
Uses the BN-RE-Conv pre-activation form for convolutional layers, except for the stem.

The bottleneck convolution does a dimensionality reduction on the number of feature maps to reduce the overall number as they accumulate across blocks.

DenseNet

Each dense block is followed by a transitional block, except for the last block.

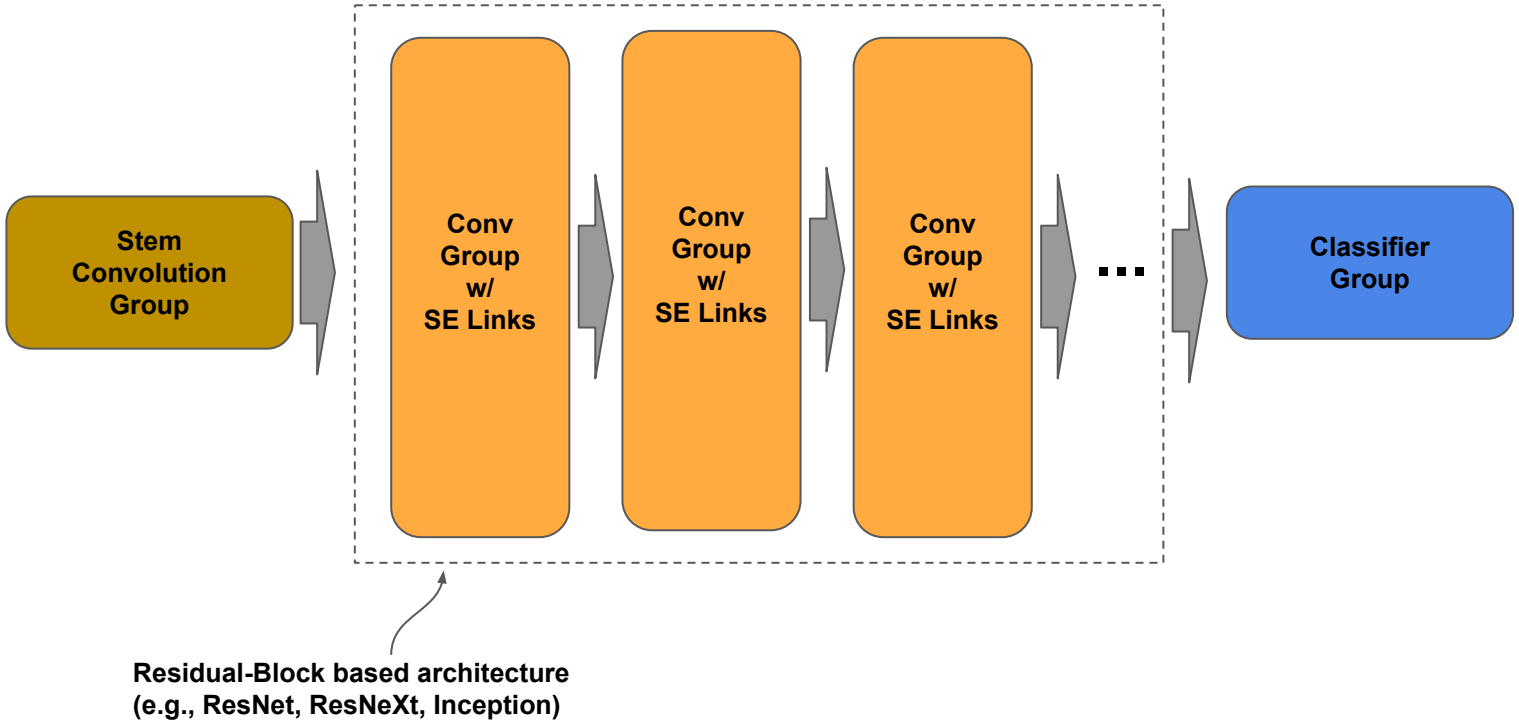
The number of feature maps is reduced by the compression metaparameter.



Finally, the compressed feature maps are further reduced by pooling.

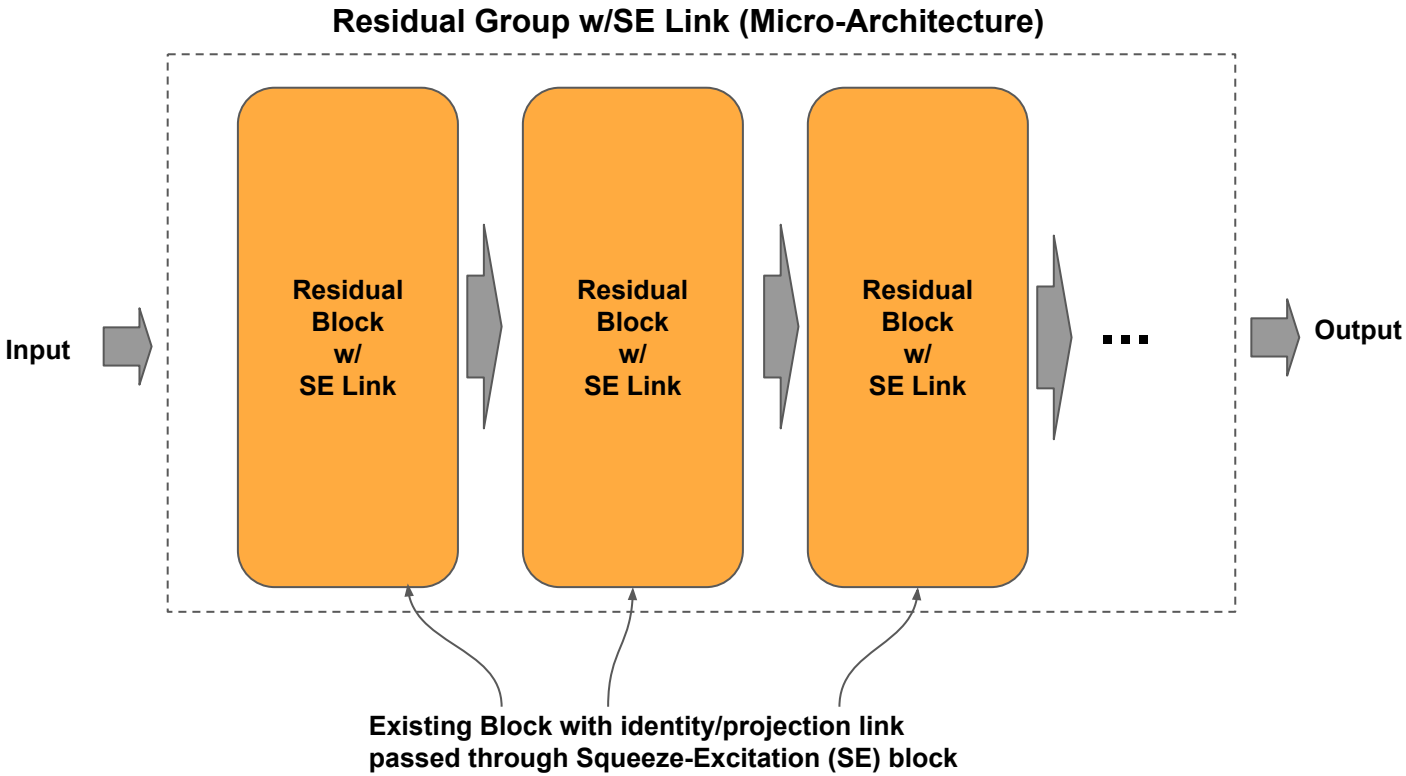
SENet

SENet Macro-Architecture



SENet is a derivative of the residual block method (ResNet/ResNeXt) by modifying the identity/projection link with a squeeze-excitation block.

SENet



The identity/projection link on each residual block is followed by a squeeze-excitation link.

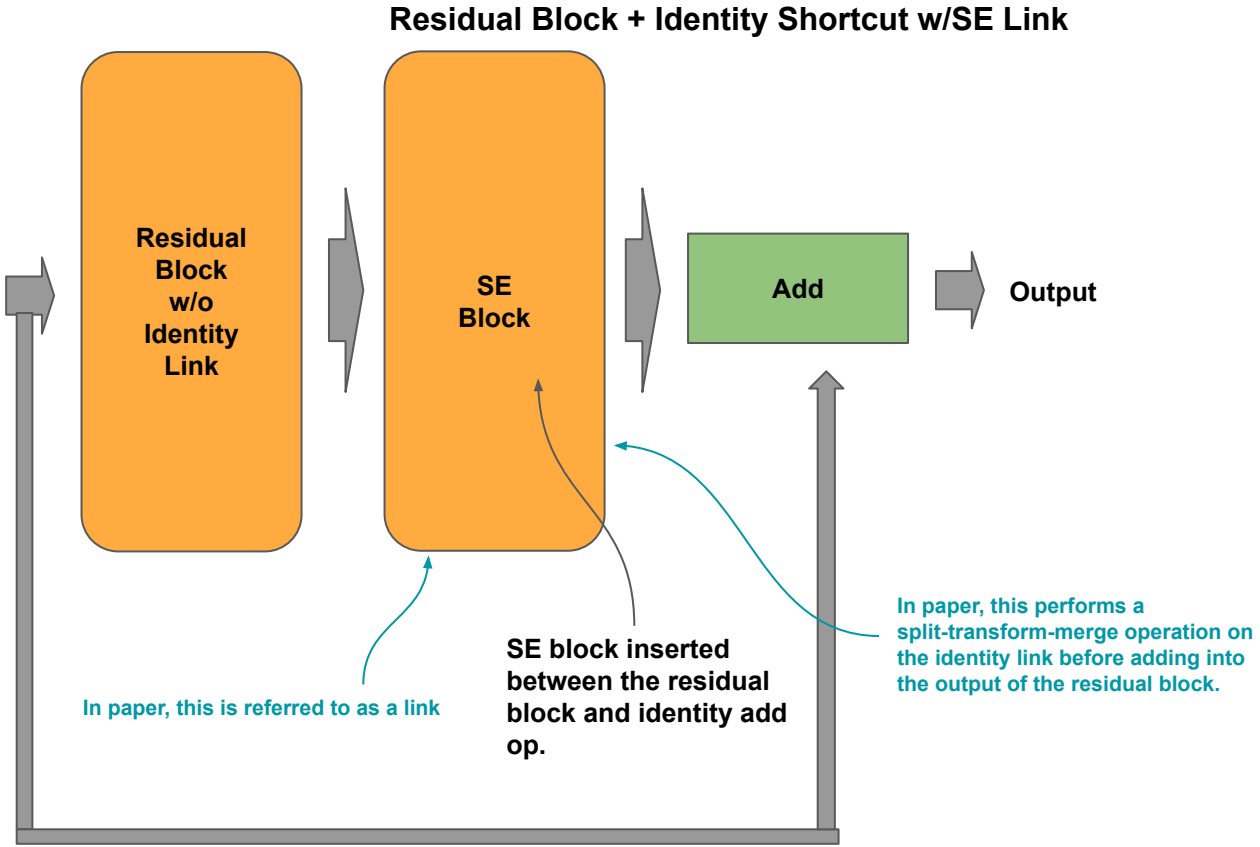
SENet

```
def group(inputs, strides=(2, 2), **metaparameters):
    # the number of blocks in the group
    blocks = metaparameters['blocks']
    # the ratio of reduction in the SE link.
    ratio = metaparameters['ratio']

    # first block is the projection shortcut block
    block = blocks.pop()
    inputs = projection_block(inputs, block['n_filters'], strides=strides, ratio=ratio)

    # remaining blocks use residual block with identity link
    for block in blocks:
        inputs = identity_block(inputs, block['n_filters'], ratio=ratio)

    return inputs
```

Metaparameters consist of blocks (which have number of filters) and the amount of reduction of filters in the squeeze-excitation link (ratio).

**code demonstration for ResNet architecture.

SENet



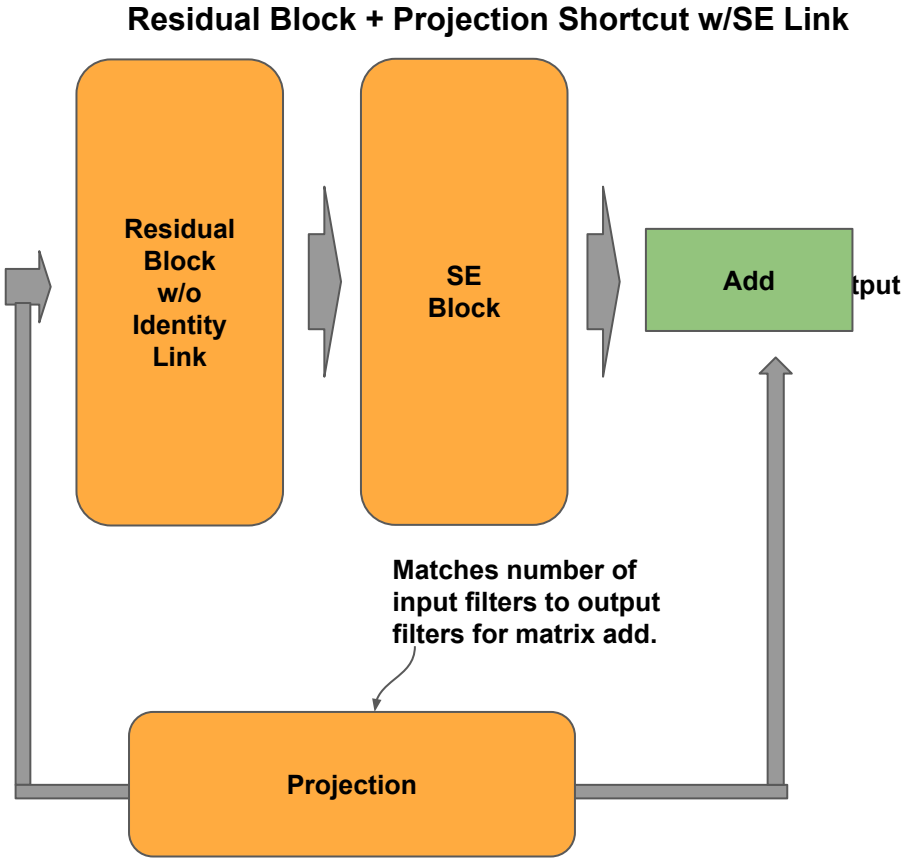
Depicts adding the Squeeze-Excitation link to the output of the residual block (prior to the add op of the identity link).

SENet

```
def identity_block(inputs, n_filters, ratio=16):
    # remember the input
    residual = inputs

    # removed for brevity ...

     # pass the output of the residual block thru the SE block
     inputs = se_block(inputs, ratio)

    # add the identity link to the output of the SE block
    inputs = Add()([residual, inputs])

    return inputs
```

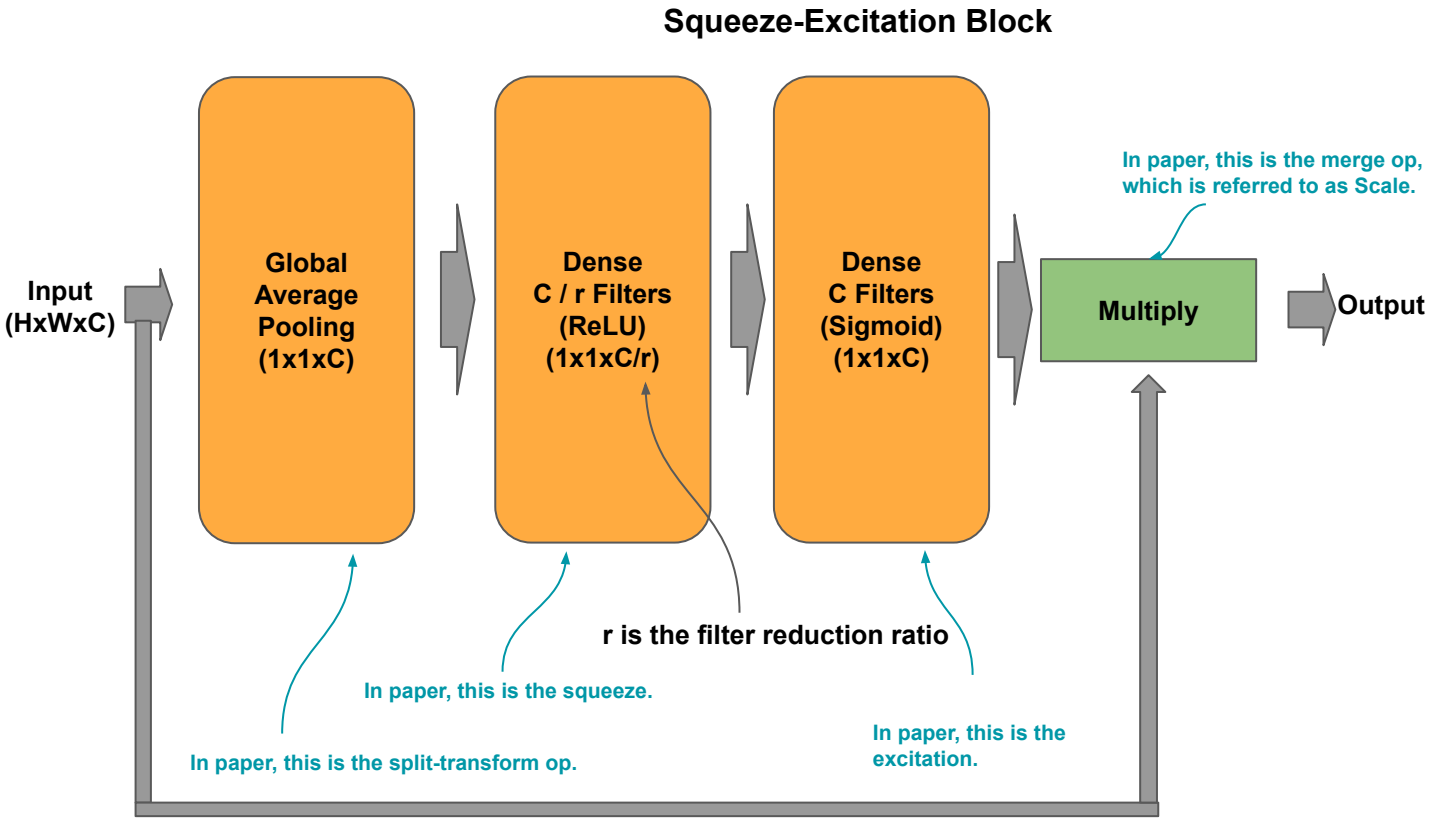
The SE block (link) is inserted between the output of the residual block and corresponding matrix add with the identity link.

SENet



Depicts adding the Squeeze-Excitation link in a residual block with a projection shortcut.

SENet



SE Link (block) uses a residual style block with identity link.

Instead of convolutions, it flattens the feature maps, and passes them thru two dense layers.

First dense layer reduce the number of feature maps (ratio) and the second restores them.

SENet

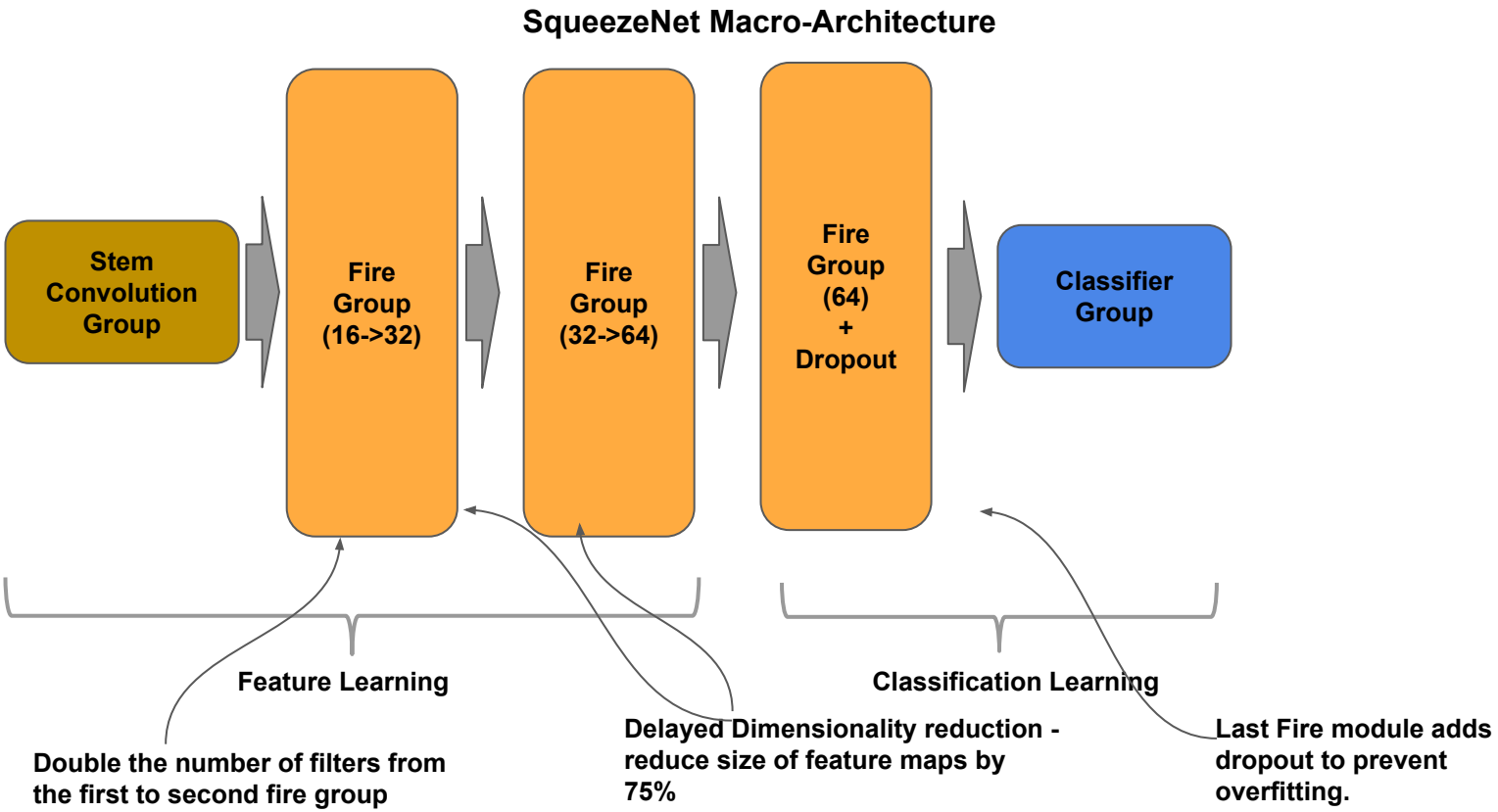
```
def se_block(inputs, n_filters, ratio=16):
    # remember the input
    residual = inputs

    # get the number of filters in the input
    n_filters = inputs.shape[-1]

    # squeeze (dimensionality reduction)
    inputs = GlobalAveragePooling2D()(inputs)

    # reshape the 1D vector into 1x1xC (C=no. of filters)
    inputs = Reshape((1, 1, n_filters))(inputs)

    # reduce the number of feature maps
    inputs = Dense(  n_filters // ratio, activation='relu')(inputs)

    # excitation (dimensionality restoration)
    inputs = Dense(filters, activation='sigmoid')(inputs)

    inputs = Multiply()([residual, inputs])
    return inputs
```

The SE block is similar in design to a residual block, in that it uses an identity link to add the input to the output of the block.

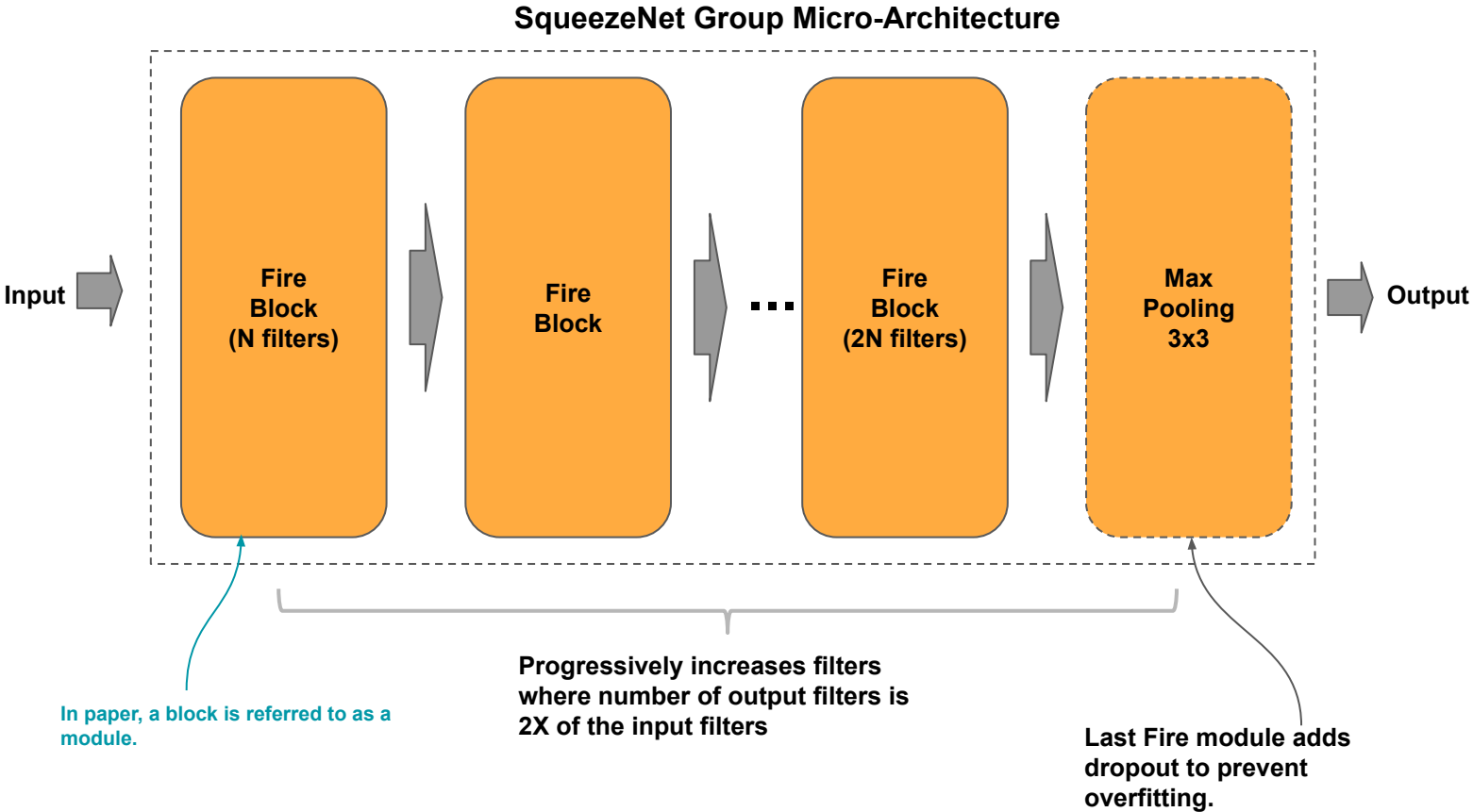
Each feature map is reduced to a single pixel, followed by reducing the number (squeeze).

The number of filters is then restored (excitation).

SqueezeNet


SqueezeNet Macro-Architecture
Stem Convolution Group
Fire Group (16->32)
Fire Group (32->64)
Fire Group (64) + Dropout
Classifier Group
Feature Learning
Classification Learning
Double the number of filters from the first to second fire group
Delayed Dimensionality reduction - reduce size of feature maps by 75%
Last Fire module adds dropout to prevent overfitting.

SqueezeNet



Each group, except the last group, delays max pooling until after the last fire block.

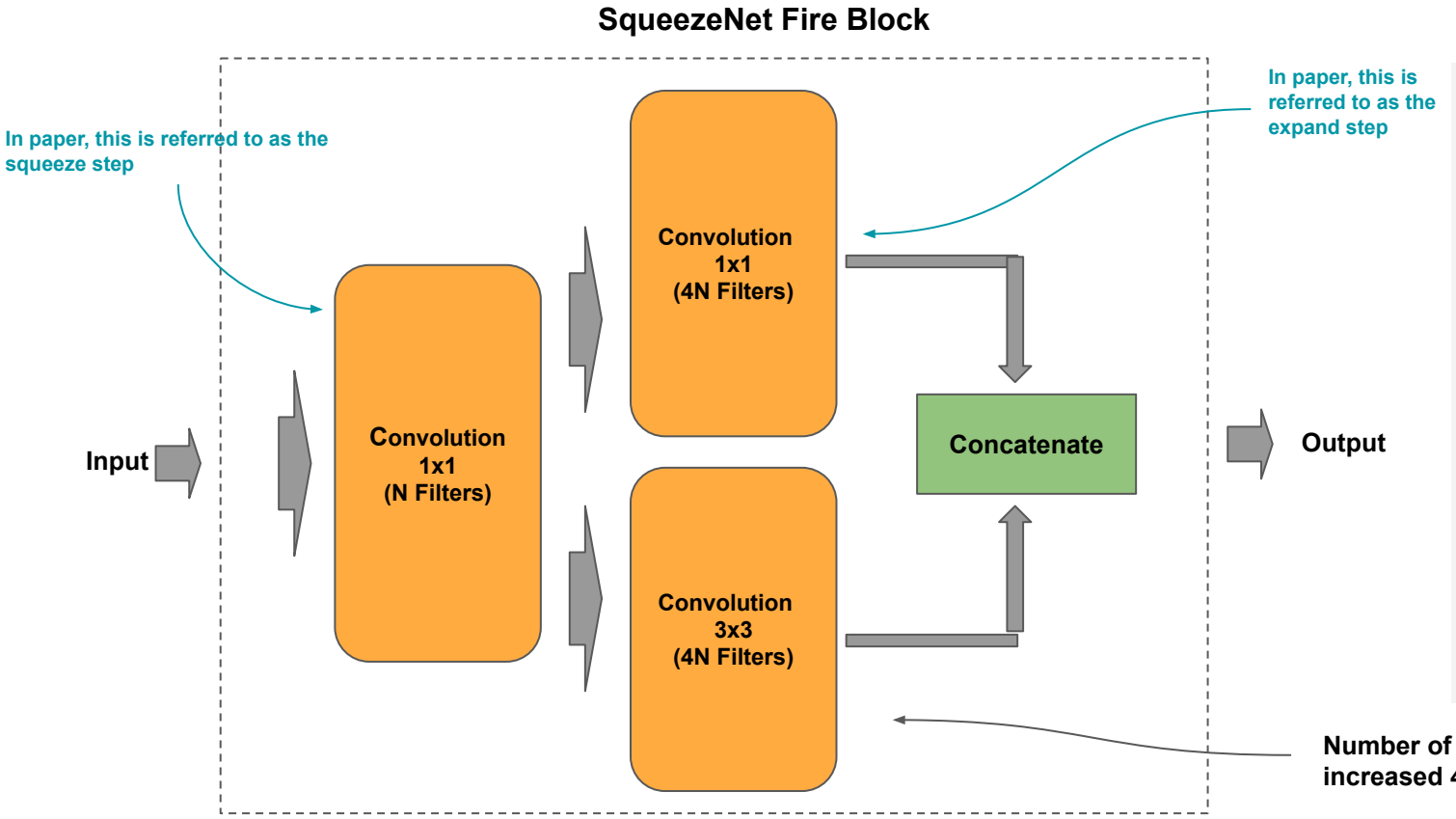
SqueezeNet

```
def group(inputs, **metaparameters):
    # the blocks for the group
    blocks = metaparameters['blocks']
    # amount of drop out (otherwise max pooling)
    dropout = metaparameters['dropout']

    for block_params in blocks:
        # number of filters for the block
        n_filters = block_params
        inputs = fire_block(inputs, n_filters)

    # delayed max pooling
    if dropout is None:
        inputs = MaxPooling2D()(inputs)
    # last group, do dropout instead
    else:
        inputs = Dropout(dropout)

    return inputs
```

The blocks metaparameter specifies the number of fire blocks, and the number of filters per block.

The last group specifies a dropout; otherwise the group ends with delayed max pooling.

SqueezeNet



The input is first squeezed by reducing the number of feature maps using a linear projection.

The features maps are then expanded by increasing the number of filters using parallel linear projection and 3x3 convolution.

SqueezeNet

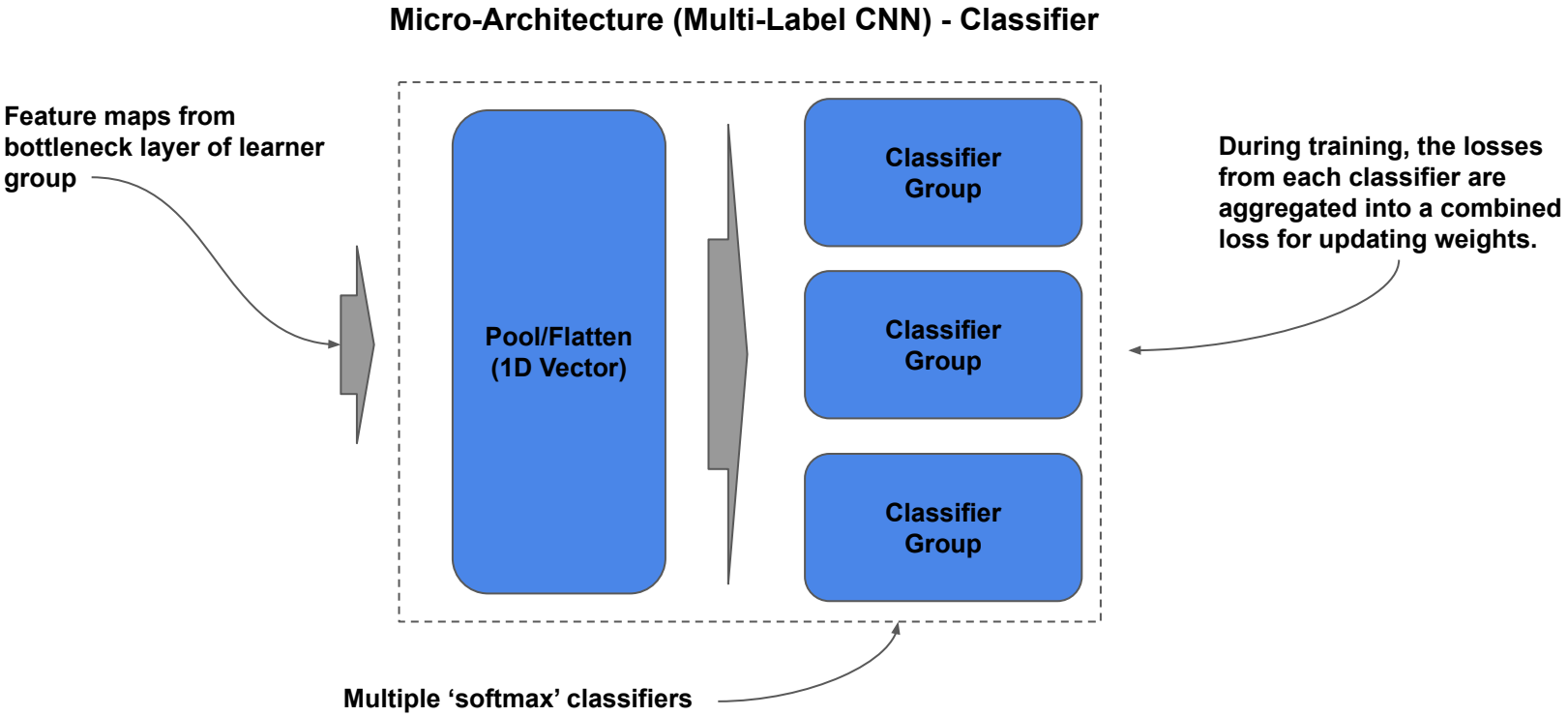
```
def fire_block(inputs, n_filters):
    # squeeze
    inputs = Conv2D(n_filters, (1, 1), strides=(1, 1), padding='same',
                    activation='relu')(inputs)

    # expand
    expand_1x1 = Conv2D(4 * n_filters, (1, 1), strides=(1, 1), padding='same',
                        activation='relu')(inputs)
    expand_3x3 = Conv2D(4 * n_filters, (1, 1), strides=(1, 1), padding='same',
                        activation='relu')(inputs)

    # concatenate the feature maps from the branches
    inputs = Concatenate()([expand_1x1, expand_3x3])

    return inputs
```

The input is reduced (squeezed) using a 1x1 linear projection convolution.

The output is then passed to two parallel convolutions to increase (expand) the number of feature maps by 4X.

The feature maps from the two parallel convolutions are concatenated for the output.

Multi-Label Classifier



Multi-Label Classifier

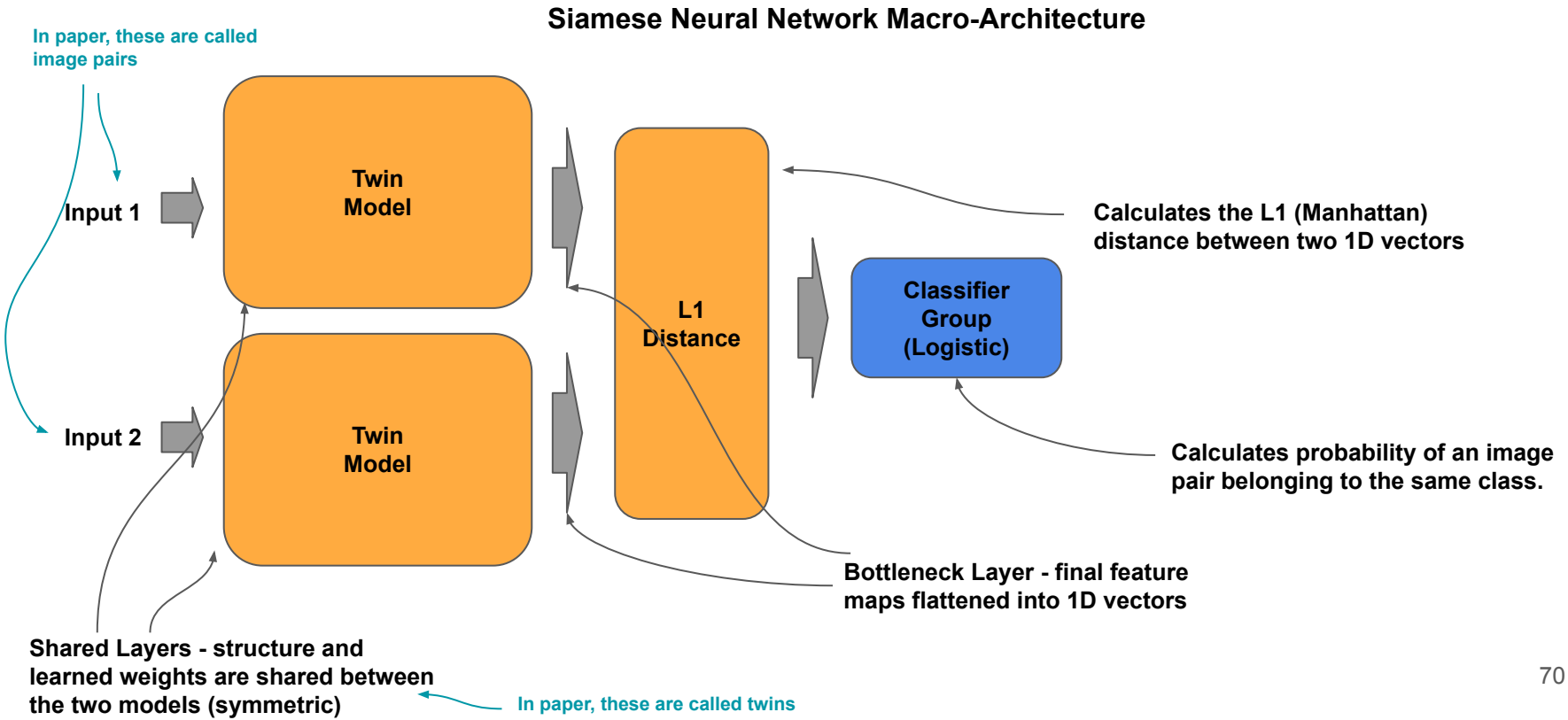
```
def classifier(x, classes):
    # Construct softmax classifier per group of classes
    outputs = []
    for n_classes in classes:
        output = Dense(n_classes, activation='softmax')(x)
        outputs.append(output)]

    # Return the multiple outputs as a list
    return outputs



# When compiling (build) the model, is equivalent to:

model = model(inputs, [outputs1, outputs2, …, outputsN])
```

A multi-label CNN is a CNN model with multiple outputs from a single input.

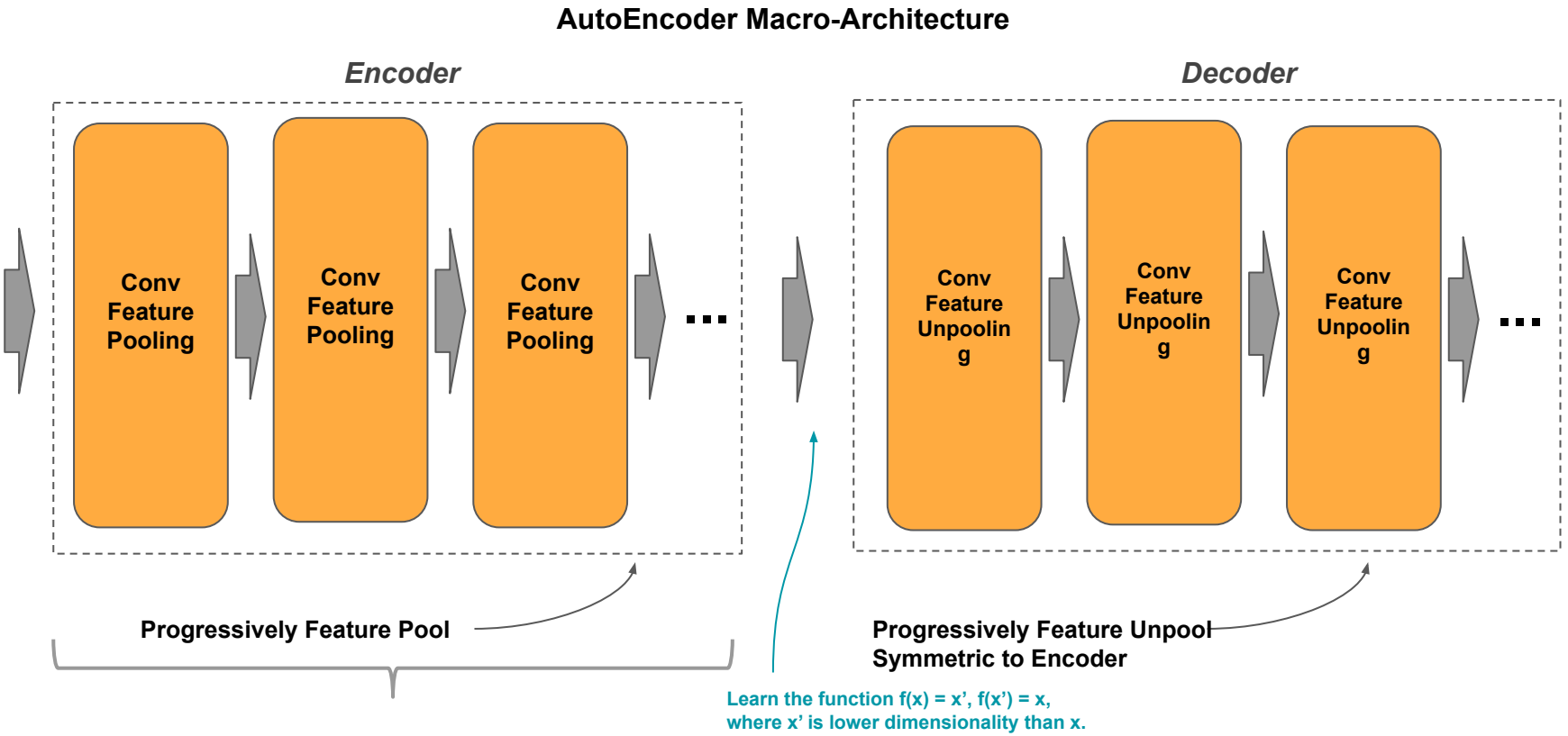
When using the Functional API, this is the same as building the model using a single input with multiple outputs.

Weights are updated during training using the combined loss across all outputs.

Siamese Twin



AutoEncoder



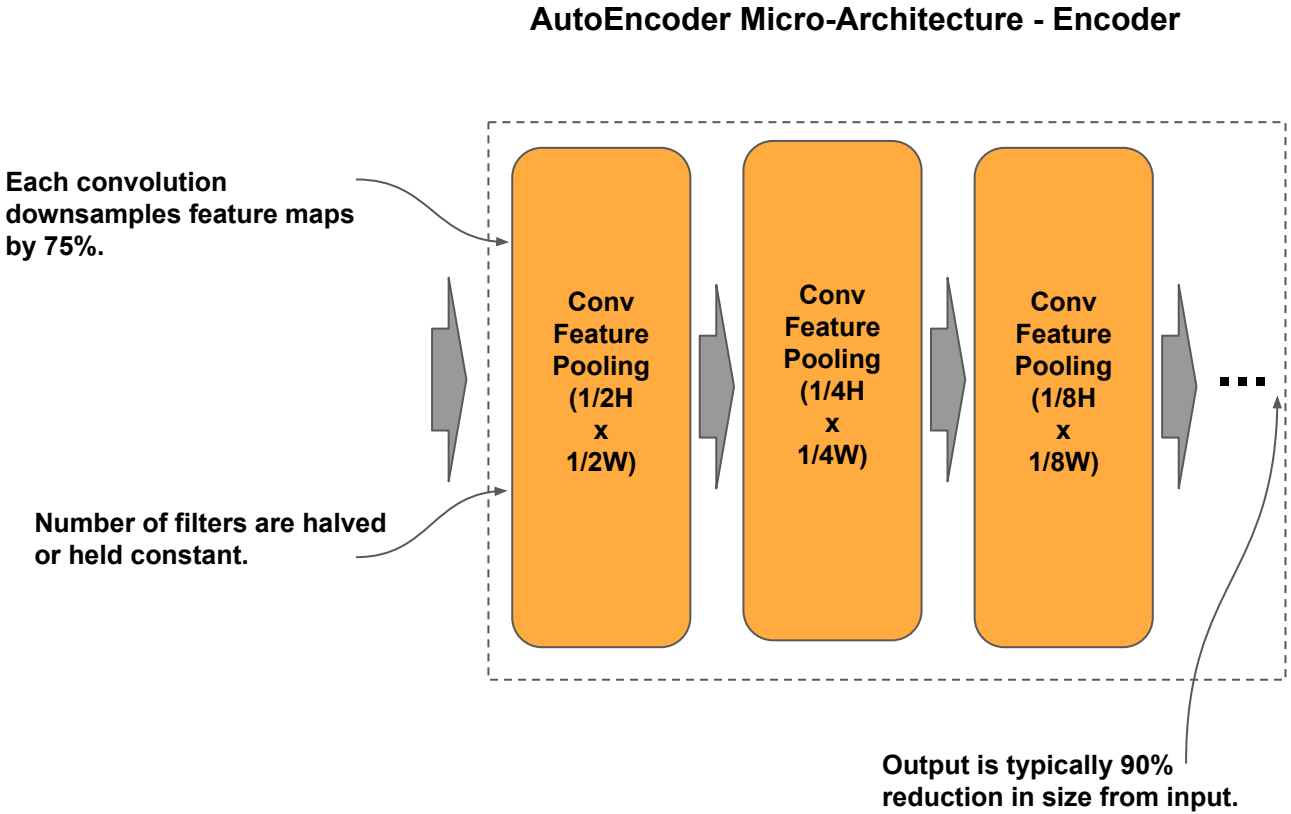
AutoEncoder

```
# metaparameter: filters per layer in encoder
layers = [ { ‘n_filters’: 64 }, { ‘n_filters’: 32 }, { ‘n_filters’: 32 } ]

# input shape to autoencoder
inputs = Input(input_shape=(32, 32, 3))

# the encoder
x = encoder(inputs, layers=layers)

# the decoder
outputs = decoder(x, layers=layers)

model = Model(inputs, outputs)

# compile using mean square error as the loss function
model.compile(loss=’mse’, ….)
```

Construct autoencoder as an encoder and then decoder, where decoder is reverse symmetric to encoder.

Loss function is mean square error.

AutoEncoder



Typically, the encoder is three layers, where each layer reduces height and width by ½.

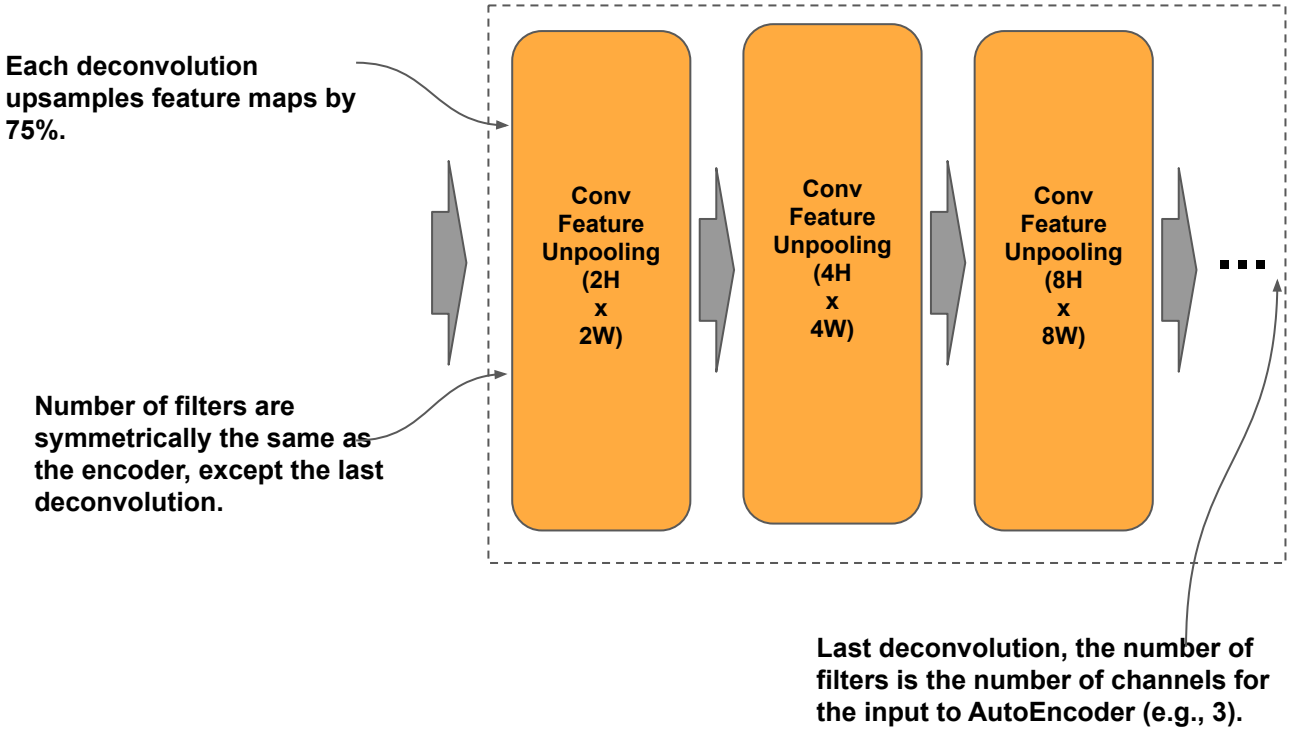
Final feature maps are ⅛ in height and width.

A high number of filters (e.g.,64) than the channels (e.g., 3) are used and progressively reduced or stay the same.

Objective is to reduce HxWx3 input by 90% for ⅛ H x ⅛ W x C

AutoEncoder

AutoEncoder Micro-Architecture - Decoder



The decoder is symmetric to the encoder, progressively increasing the feature map sizes in reverse order.

The last deconvolution restores the number of channels to the input.

AutoEncoder

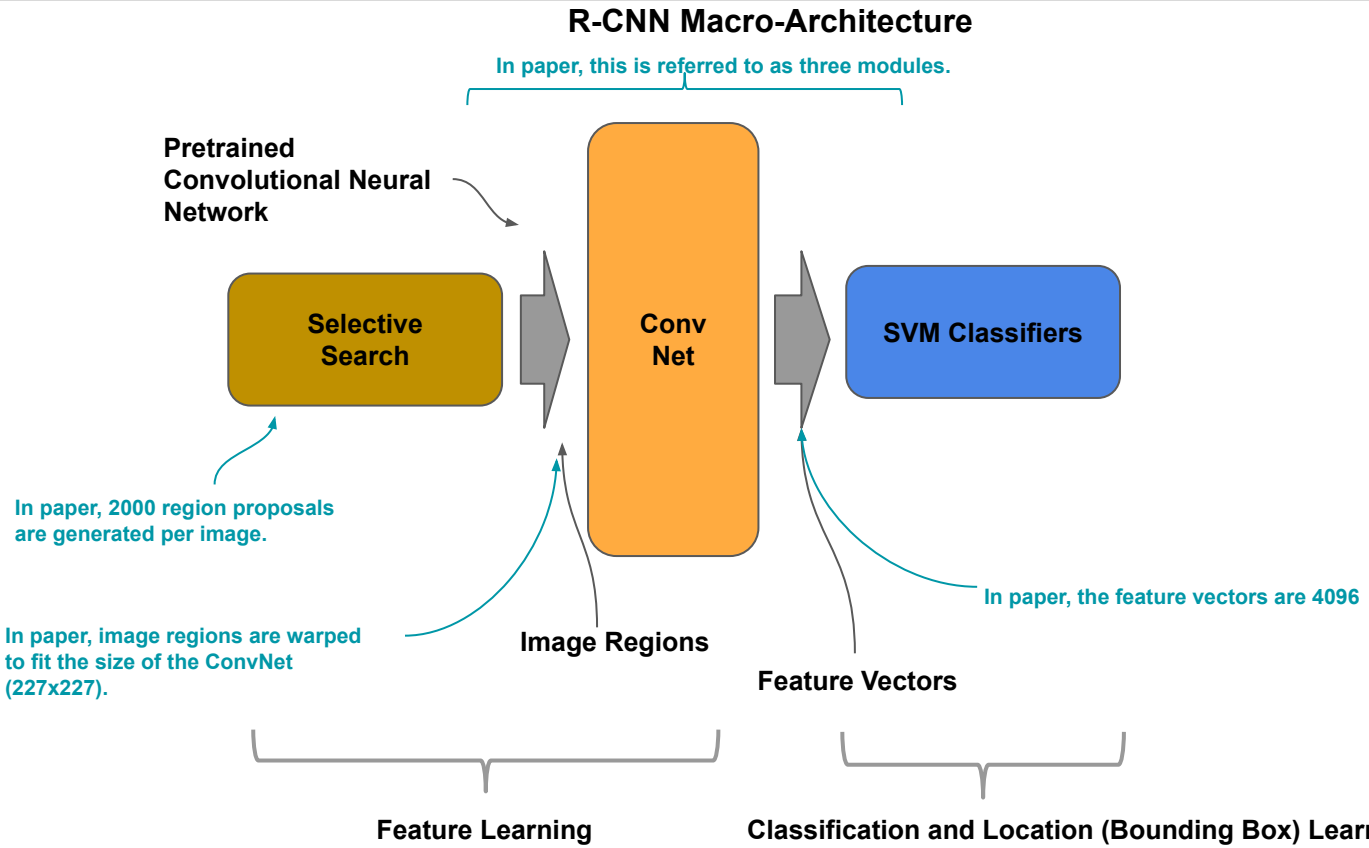
```
def encoder(x, **metaparameters):
      # Progressive feature pooling
      layers = metaparameters['layers']
      for layer in layers:
          n_filters = layer['n_filters']
          x = Conv2D(n_filters, (3, 3), strides=2, padding='same')(x)

      # Return the bottleneck layer (encoding)
      return x

 def decoder(x, **metaparameters):
     # Progressive feature unpooling (in reverse order)
     layers = metaparameters['layers']
     for _ in range(len(layers-1), 0, -1):
          n_filters = layers[_]['n_filters']
          x = Conv2DTranspose(n_filters, (3, 3), strides=2)(x)

     # On last deconvolution, restore the number of channels to the input.
     x = Conv2DTranspose(3, (3, 3), strides=2, padding='same')(x)
```

Progressively reduce the feature map HxW by ½ (strides=2).

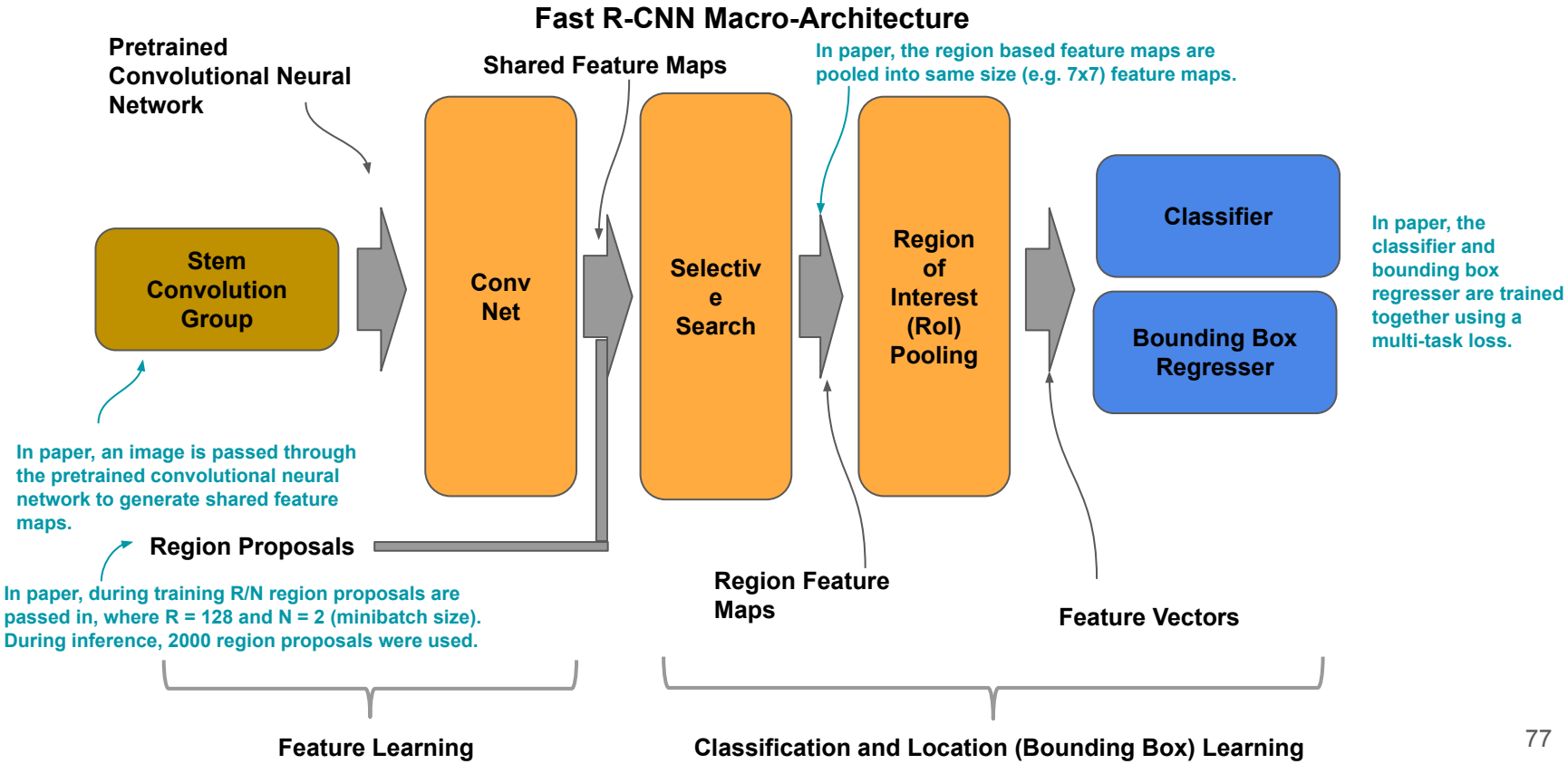
Generally, the number of filters is halved or kept the same.

Progressively increase the feature map HxW by 2 (strides=2).

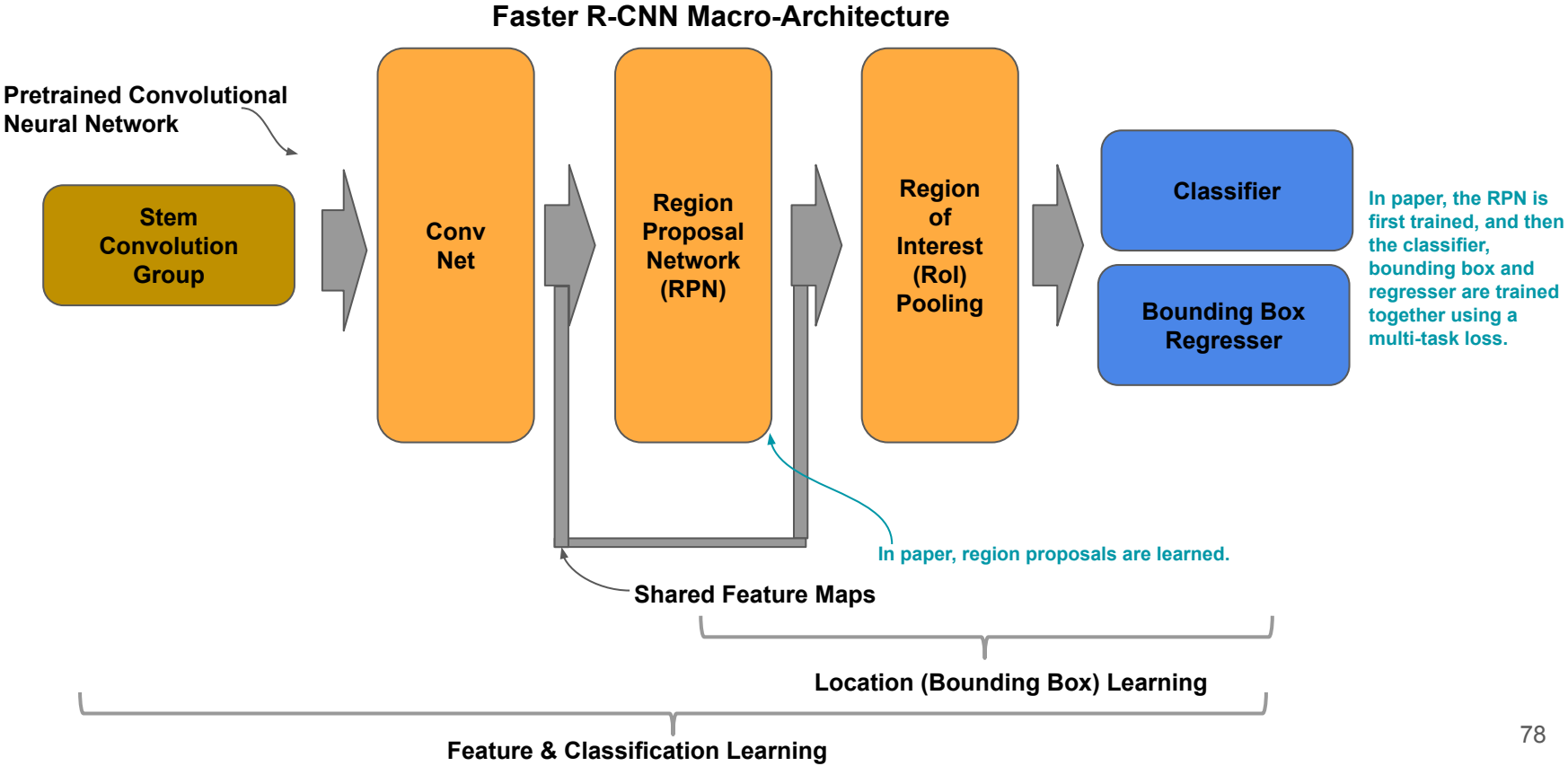
R-CNN


R-CNN Macro-Architecture
In paper, this is referred to as three modules.
Pretrained Convolutional Neural Network
Selective Search
Conv Net
SVM Classifiers
In paper, 2000 region proposals are generated per image.
In paper, image regions are warped to fit the size of the ConvNet (227x227).
Image Regions
Feature Vectors
In paper, the feature vectors are 4096
Feature Learning
Classification and Location (Bounding Box) Learning

Fast R-CNN


Fast R-CNN Macro-Architecture
Pretrained Convolutional Neural Network
Shared Feature Maps
In paper, the region based feature maps are pooled into same size (e.g. 7x7) feature maps.
Stem Convolution Group
Conv Net
Selectiv e Search
Region of Interest (RoI) Pooling
Classifier
Bounding Box Regresser
In paper, the classifier and bounding box regresser are trained together using a multi-task loss.
In paper, an image is passed through the pretrained convolutional neural network to generate shared feature maps.
Region Proposals
In paper, during training R/N region proposals are passed in, where R = 128 and N = 2 (minibatch size). During inference, 2000 region proposals were used.
Region Feature Maps
Feature Vectors
Feature Learning
Classification and Location (Bounding Box) Learning

Faster R-CNN

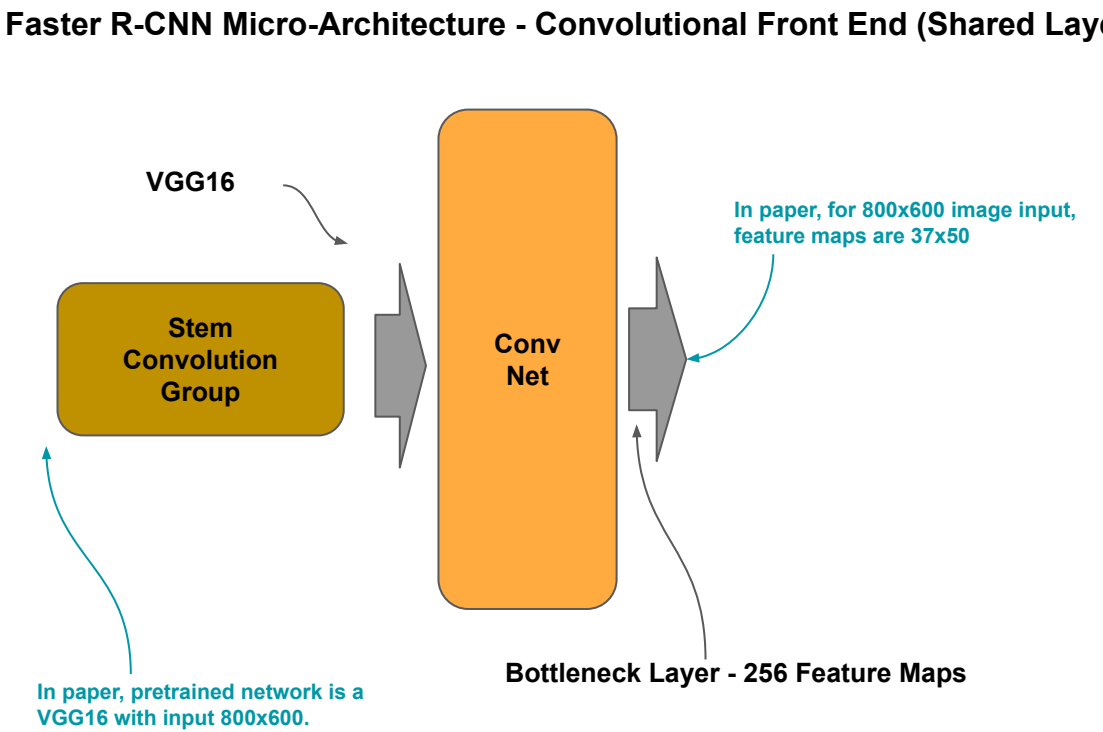

Faster R-CNN Macro-Architecture
Pretrained Convolutional Neural Network
Stem Convolution Group
Conv Net
Region Proposal Network (RPN)
Region of Interest (RoI) Pooling
Classifier
Bounding Box Regresser
In paper, the RPN is first trained, and then the classifier, bounding box and regresser are trained together using a multi-task loss.
In paper, region proposals are learned.
Shared Feature Maps
Location (Bounding Box) Learning
Feature & Classification Learning

Faster R-CNN

```
def stem(inputs):
    # VGG16 w/o classifier ...
    return bottleneck_feature_maps

def learner(x):
    # region proposal network (RPN) ...

    # region of interest pooling (RoI) ...

    return pooled_roi_feature_maps

def classifier(x):
    # softmax classifier for object classification ...
    # linear regression for bounding box ...

    return object_classes,  bounding_boxes
```

TODO

Faster R-CNN


Faster R-CNN Micro-Architecture - Convolutional Front End (Shared Layers)
VGG16
Stem Convolution Group
Conv Net
In paper, for 800x600 image input, feature maps are 37x50
In paper, pretrained network is a VGG16 with input 800x600.
Bottleneck Layer - 256 Feature Maps