Transfer learning

Keras & TensorFlow

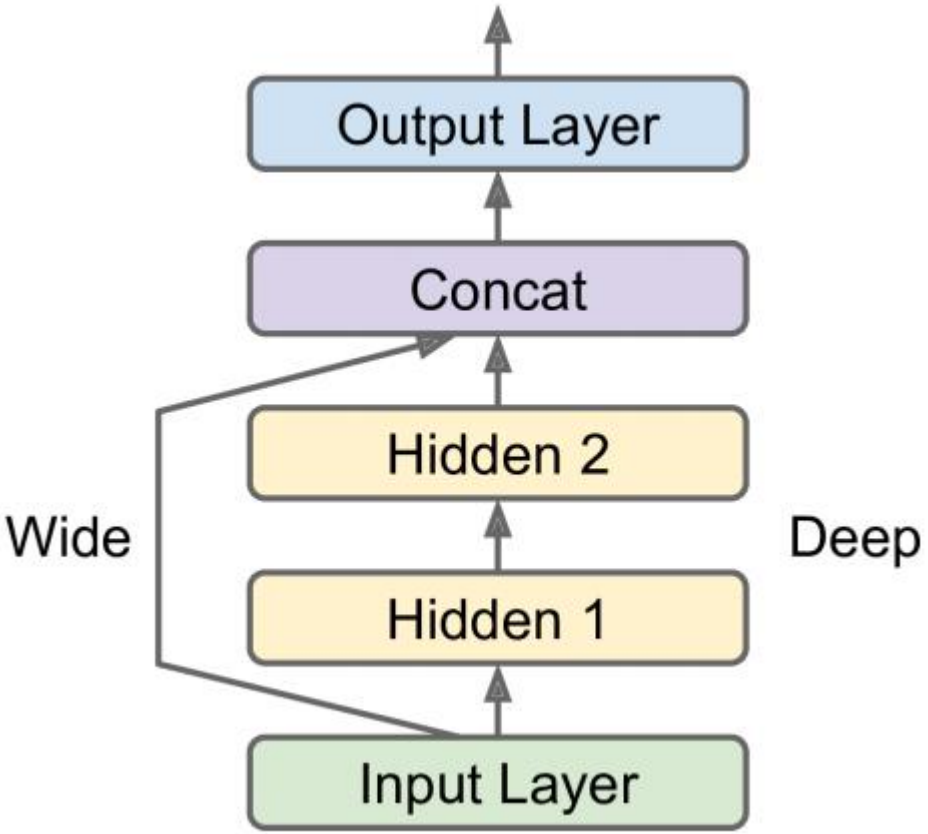
Hichem Felouat
hichemfel@gmail.com

Transfer learning

Transfer learning (TL) is a research problem in ML that focuses on storing knowledge gained while solving one problem and applying it to a different but related problem. **For example**, knowledge gained while learning to recognize Cats could apply when trying to recognize Tigers.

Building Complex Models - Functional API

- The Keras functional API is a way to create models that is more **flexible** than **the tf.keras.Sequential** API.
- The functional API can handle models with **non-linear topology**, models with **shared layers**, and models with **multiple inputs or outputs**.
- This architecture makes it possible for the neural network to learn both **deep patterns** (using the deep path) and **simple rules** (through the short path).



Building Complex Models - Functional API

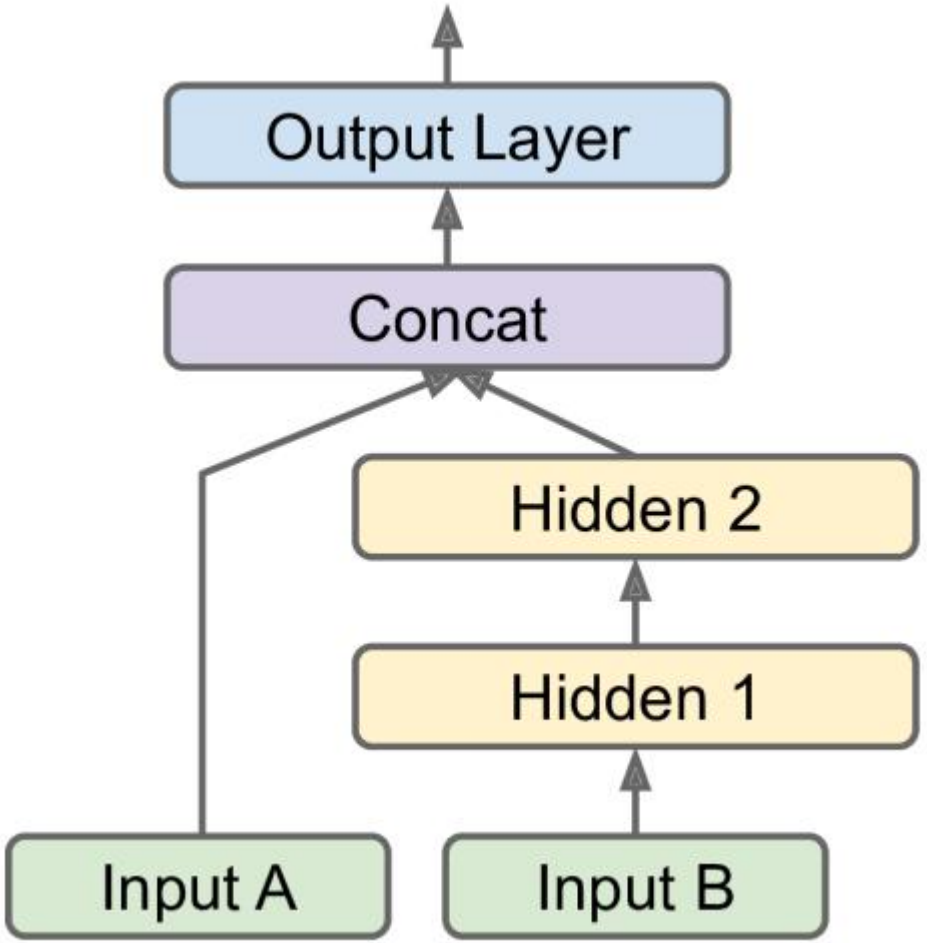
```
input_   = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu")(input_)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat   = keras.layers.Concatenate()([input_, hidden2])
output   = keras.layers.Dense(1)(concat)
model    = keras.Model(inputs=[input_], outputs=[output])
```

Once you have built the Keras model, everything is exactly like earlier, so there's no need to repeat it here: you must compile the model, train it, evaluate it, and use it to make predictions.

Building Complex Models - Functional API

What if you want to send a subset of the features through the wide path and a different subset (**possibly overlapping**) through the deep path?!

- **In this case, one solution is to use multiple inputs.**



Building Complex Models - Functional API

For example, suppose we want to send five features through the wide path (features 0 to 4), and six features through the deep path (features 2 to 7) :

```
input_A  = keras.layers.Input(shape=[5], name="wide_input")
input_B  = keras.layers.Input(shape=[6], name="deep_input")
hidden1 = keras.layers.Dense(30, activation="relu")(input_B)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat   = keras.layers.concatenate([input_A, hidden2])
output    = keras.layers.Dense(1, name="output")(concat)
model    = keras.Model(inputs=[input_A, input_B], outputs=[output])
```

- You should name at least the most important layers.

Building Complex Models - Functional API

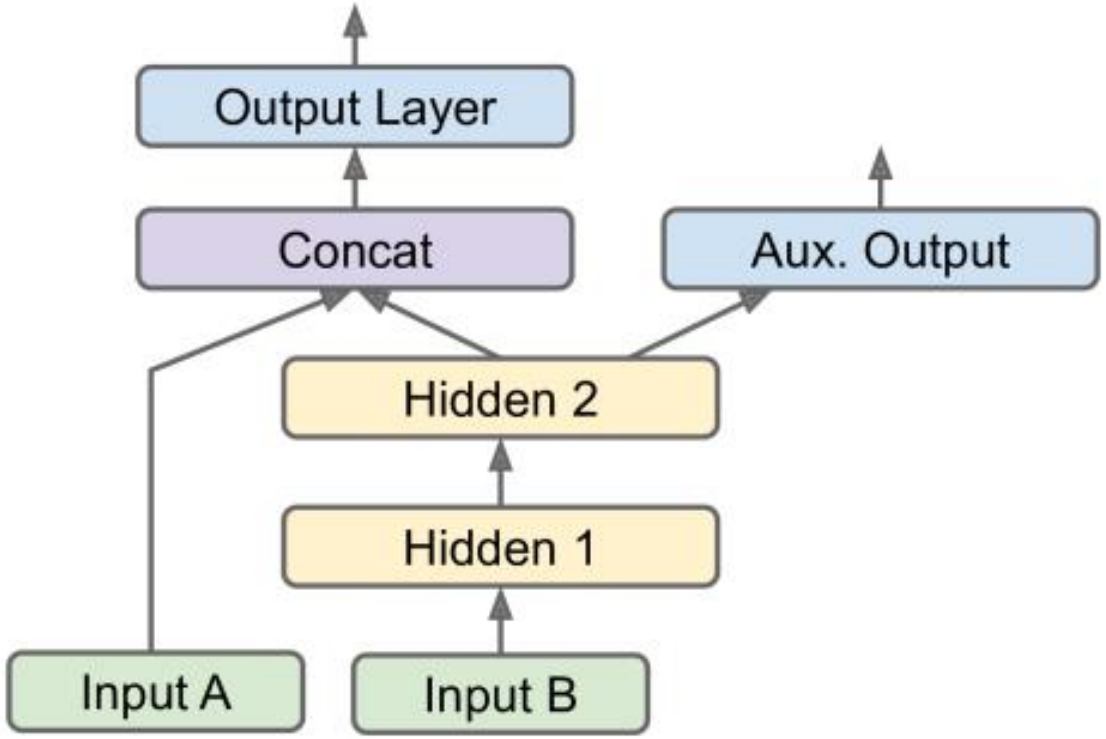
We can compile the model as usual, but when we call the **fit() method**, instead of passing a single input matrix X_train, **we must pass a pair of matrices (X_train_A, X_train_B)**: one per input. The same is true for **X_valid**, and also for **X_test** and **X_new** when you call evaluate() or predict():

```
history = model.fit((X_train_A, X_train_B), y_train, epochs=20,
                    validation_data = ((X_valid_A, X_valid_B), y_valid))
model_evaluate = model.evaluate((X_test_A, X_test_B), y_test)
y_pred = model.predict((X_new_A, X_new_B))
```

Building Complex Models - Functional API

Another use case is as a **regularization technique** (i.e., a training constraint whose objective is to reduce overfitting and thus improve the model's ability to generalize).

For example, you may want to add some auxiliary outputs in a neural network architecture to ensure that the underlying part of the network learns something useful on its own, without relying on the rest of the network.



Building Complex Models - Functional API

```
[...] # Same as before, up to the main output layer

output       = keras.layers.Dense(1, name="main_output")(concat)
aux_output = keras.layers.Dense(1, name="aux_output")(hidden2)
model = keras.Model(inputs=[input_A, input_B], outputs=[output,
                                                        aux_output])
```

- Each output will need its own loss function. Therefore, when we compile the model, we should pass a list of losses.

```
model.compile(loss=["mse", "mse"], loss_weights=[0.9, 0.1], optimizer="sgd")
```

Building Complex Models - Functional API

- We care much more about the main output than about the auxiliary output, so we want to give the main output's loss a much greater weight.
- we need to provide labels for each output (In this example, we used the same labels).

```
history = model.fit([X_train_A, X_train_B], [y_train, y_train], epochs=20,
                    validation_data=([X_valid_A, X_valid_B], [y_valid, y_valid]))
```

- When we evaluate the model, Keras will return the total loss, as well as all the individual losses:

```
total_loss, main_loss, aux_loss = model.evaluate([X_test_A, X_test_B],
                                                 [y_test, y_test])

y_pred_main, y_pred_aux = model.predict([X_new_A, X_new_B])
```

Reusing Pretrained Layers

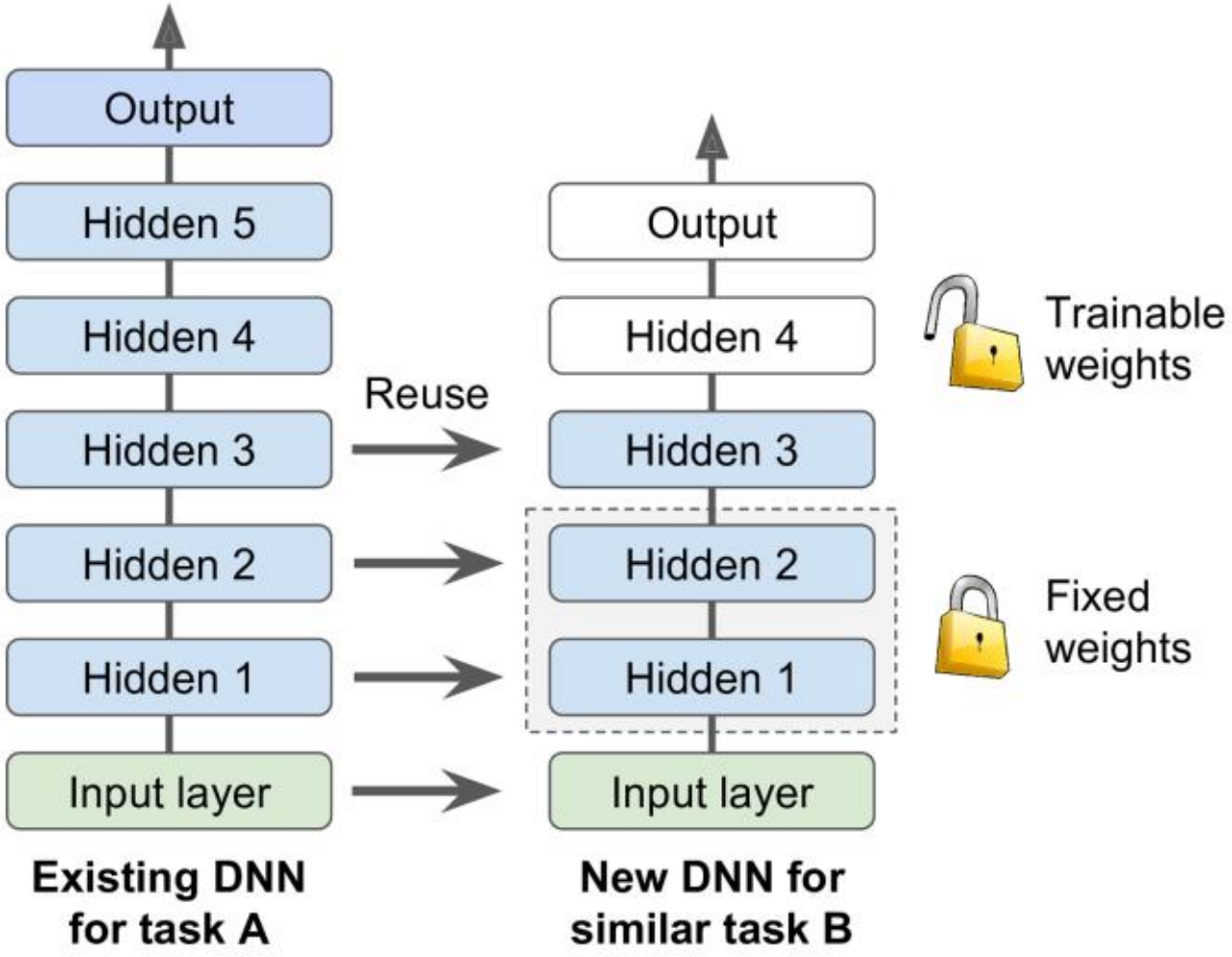
- It is generally not a good idea to train a very large DNN from scratch: instead, you should always try to find an existing neural network that accomplishes a similar task to the one you are trying to tackle then reuse the lower layers of this network.

- This technique is called **transfer learning**.

- It will not only **speed up training** considerably but also require significantly **less training data**.

- The output layer of the original model **should usually be replaced** because it is most likely not useful at all for the new task, and it may not even have the right number of outputs for the new task.

Reusing Pretrained Layers



- ***Transfer learning will work best when the inputs have similar low-level features.***

Reusing Pretrained Layers

- Try freezing all the reused layers first (i.e., make their weights non-trainable so that Gradient Descent won't modify them), then train your model and see how it performs. Then try unfreezing one or two of the top hidden layers to let backpropagation tweak them and see if performance improves.

```
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))

# If you want to avoid affecting model_A
model_A_clone = keras.models.clone_model(model_A)
model_A_clone.set_weights(model_A.get_weights())
```

Reusing Pretrained Layers

- The new output layer was initialized randomly it will make large errors.
- Freeze the reused layers during the first few epochs, giving the new layer some time to learn reasonable weights.

```
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False

model_B_on_A.compile( ... )
history = model_B_on_A.fit( ..., epochs = 5, ... )

for layer in model_B_on_A.layers[:-1]:
    layer.trainable = True

model_B_on_A.compile( ... )
history = model_B_on_A.fit( ... )
model_B_on_A.evaluate( ... )
```

Reusing Pretrained Layers

Pretraining on an Auxiliary Task

If you do not have much labeled training data, one last option is to **train a first neural network on an auxiliary task** for which you can easily obtain or generate labeled training data, then **reuse the lower layers of that network for your actual task**. The first neural network's lower layers will learn feature detectors that will likely be reusable by the second neural network.

CNN Variations

Over the years, variants of the **CNN architecture** have been developed, leading to amazing advances in the field. A good measure of this progress is the error rate in competitions such as the ILSVRC **ImageNet challenge**.

1. LeNet-5 (1998)
2. AlexNet (2012)
3. VGG-16 (2014)
4. Inception-v1
5. Inception-v3
6. ResNet-50
7. Xception (2016)
8. Inception-v4 (2016)
9. Inception-ResNets
10. ResNeXt-50 (2017)

https://towardsdatascience.com/illustrated-10-cnn-architectures-95d78ace614d

CNN Variations - Legend

Layers

conv 3×3 — Convolutional operations, in **red**

avg-pool 2×2 — Pooling operations, in **grey**

concat — Merge operations eg. concat, add in **purple**

Dense layer, **blue**

Activation Functions

T — Tanh

R — ReLU

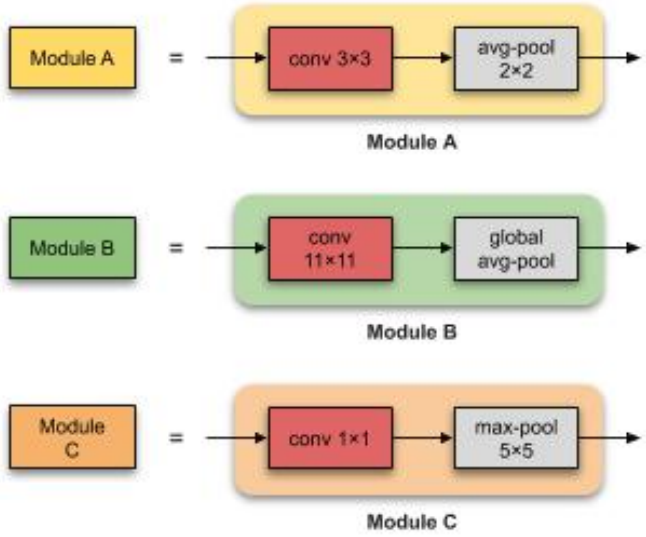
Other Functions

B — Batch normalisation

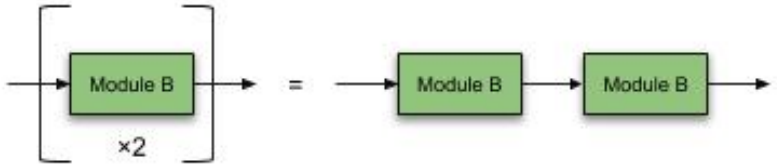
S — Softmax

Modules/Blocks

Modules (groups of convolutional, pooling and merge operations), in **yellow, green,** or **orange**.
The operations that make up these modules will also be shown.



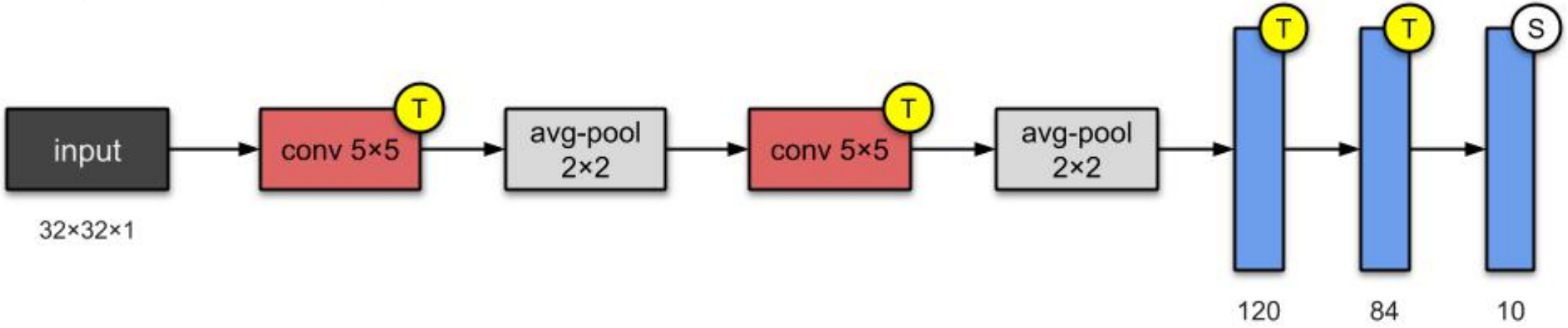
Repeated layers or modules/blocks



CNN Variations - LeNet-5

- It was created by **Yann LeCun in 1998** and has been widely used for **handwritten digit recognition (MNIST)**.

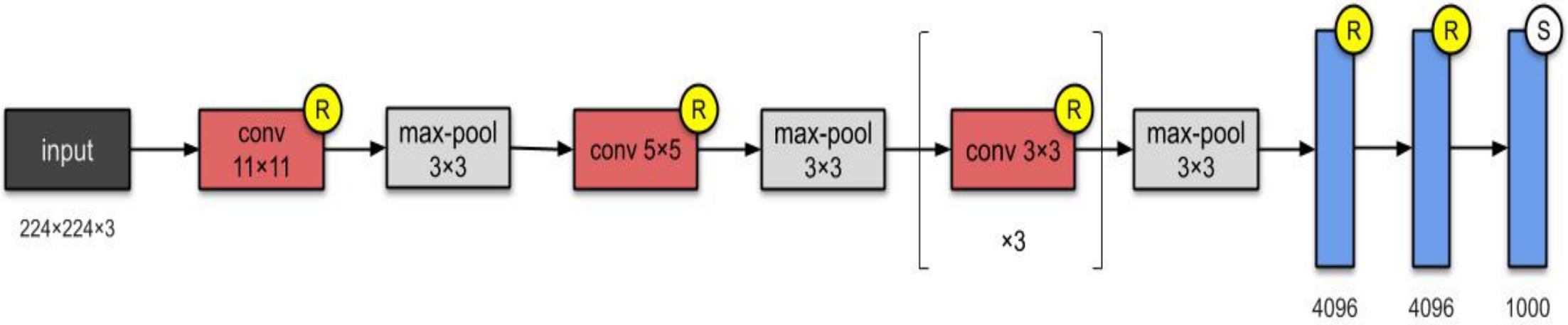
- This architecture has become the standard **"template"**: stacking convolutions and pooling layers and ending the network with one or more fully-connected layers.



CNN Variations - AlexNet

- The AlexNet CNN architecture won the 2012 ImageNet ILSVRC challenge.

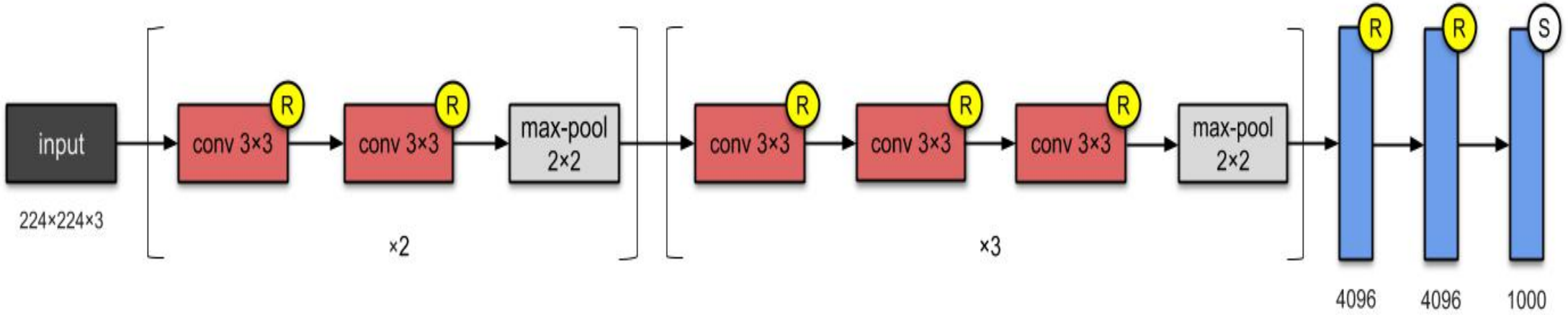
- To reduce overfitting, the authors used two regularization techniques. First, they applied **dropout** with a 50% dropout rate during training to the outputs of layers F1 and F2. Second, they performed **data augmentation**.



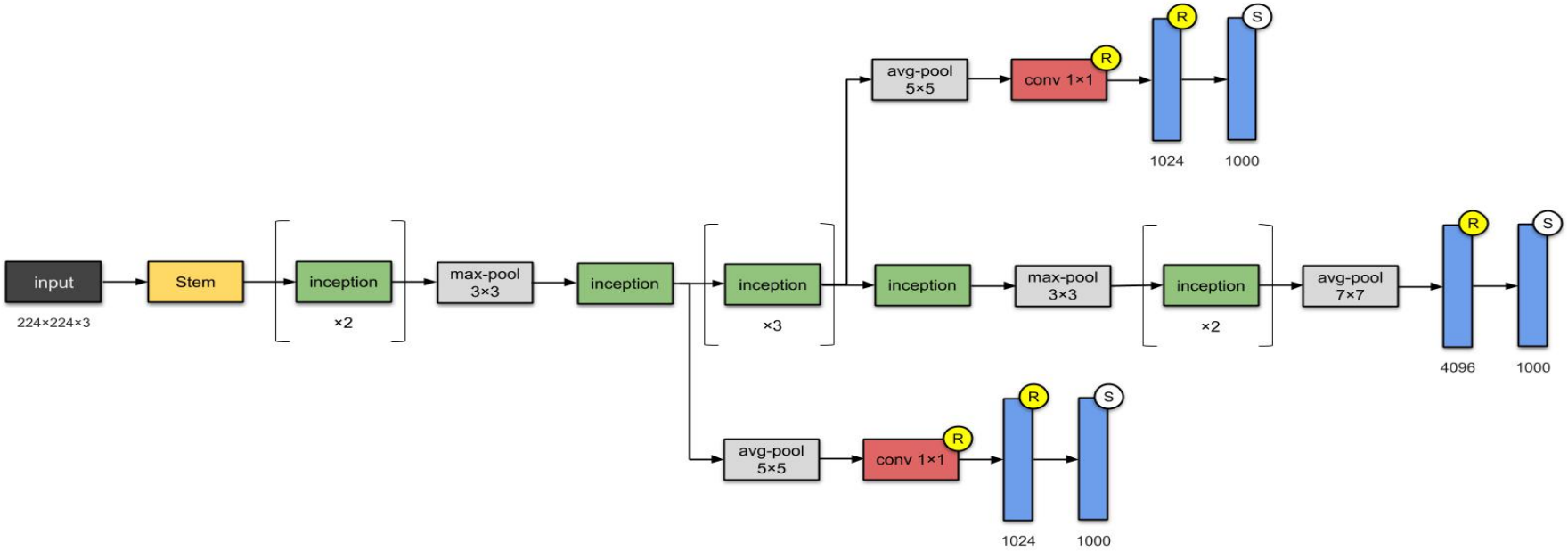
CNN Variations - VGG

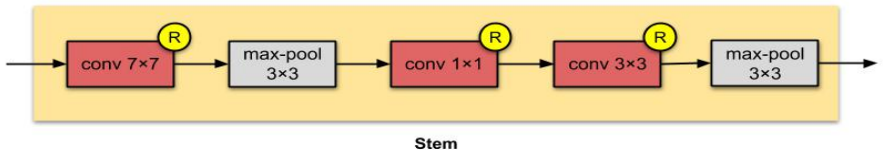
- The runner-up in the ILSVRC 2014 challenge was VGG, developed by Karen Simonyan and Andrew Zisserman from the Visual Geometry Group (VGG) research lab at Oxford University.

- VGG-16 architecture and VGG-19 architecture

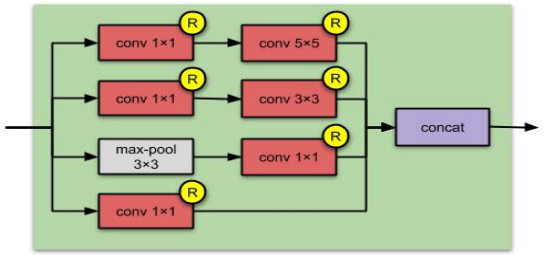


CNN Variations - GoogLeNet (Inception)





Stem

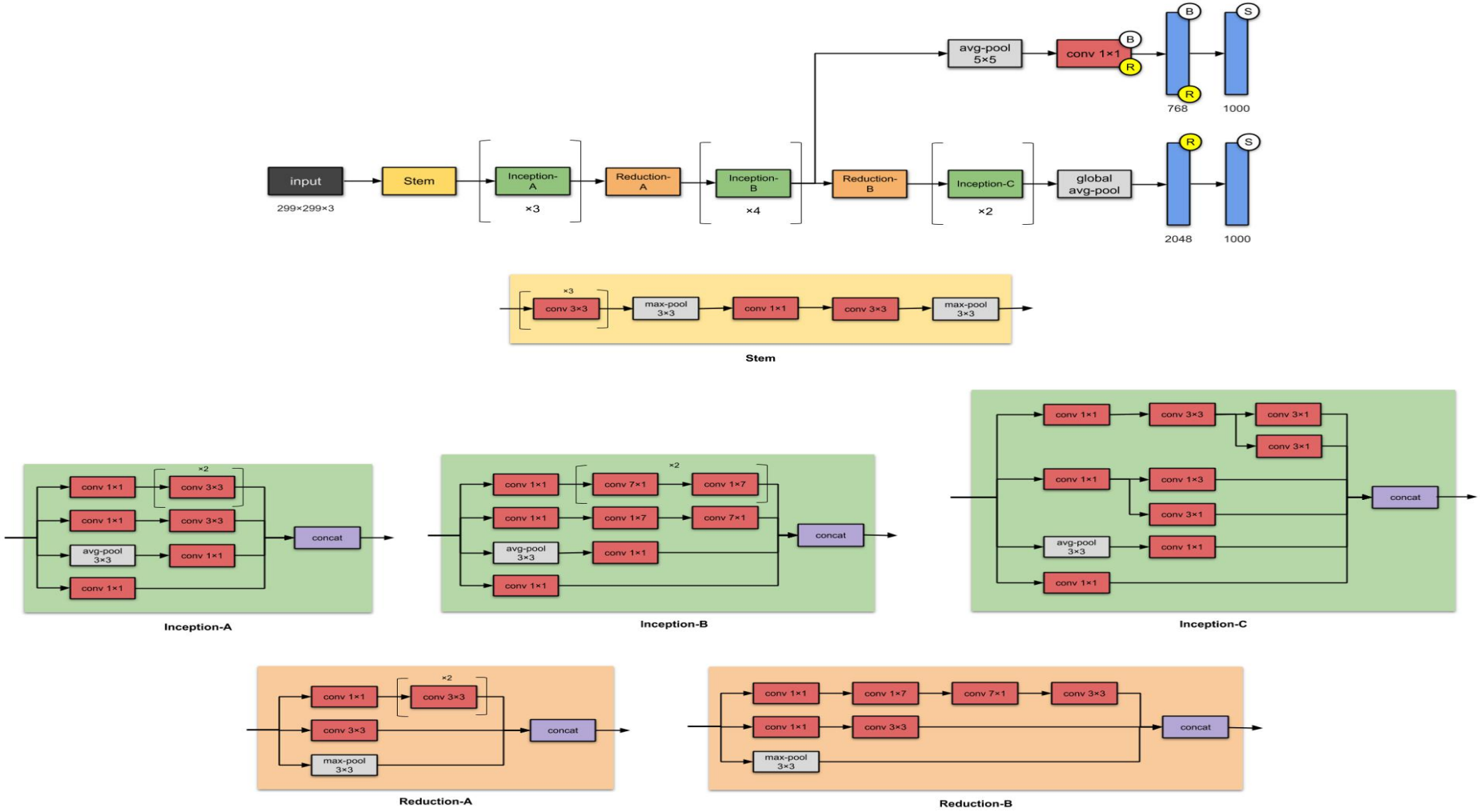


Inception module

CNN Variations - GoogLeNet (Inception)

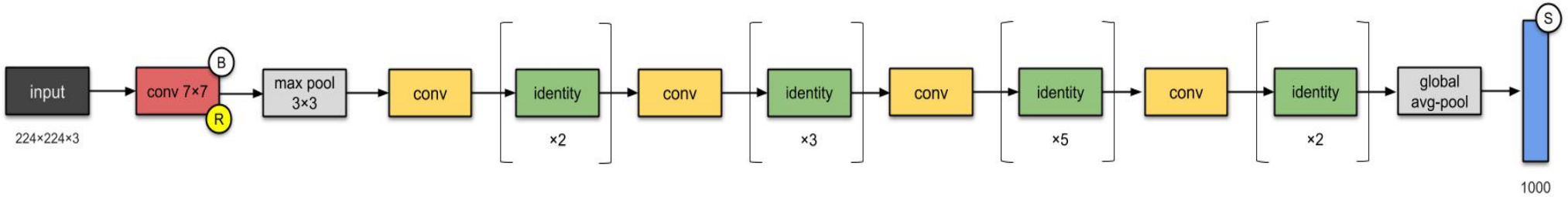
- Inception modules allow GoogLeNet to use parameters much more efficiently than previous architectures.
- This 22-layer architecture with 5M parameters is called the Inception-v1.
- Having parallel towers of convolutions with different filters, followed by concatenation, captures different features at 1×1, 3×3 and 5×5, thereby "clustering" them.
- 1×1 convolutions are used for dimensionality reduction to remove computational bottlenecks.
- Due to the activation function from 1×1 convolution, its addition also adds nonlinearity.

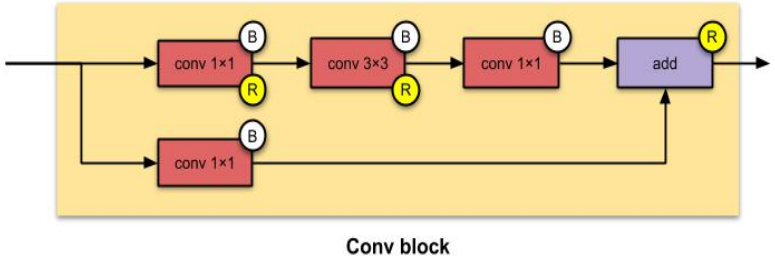
CNN Variations - GoogLeNet (Inception V3)



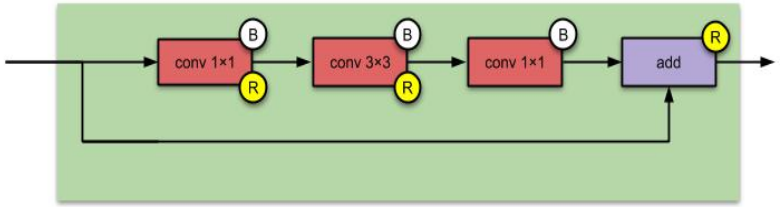
CNN Variations - ResNet-50

- The basic building block for ResNets are the conv and identity blocks.
- It uses skip connections (also called shortcut connections).
- If you add many skip connections, the network can start making progress even if several layers have not started learning yet.



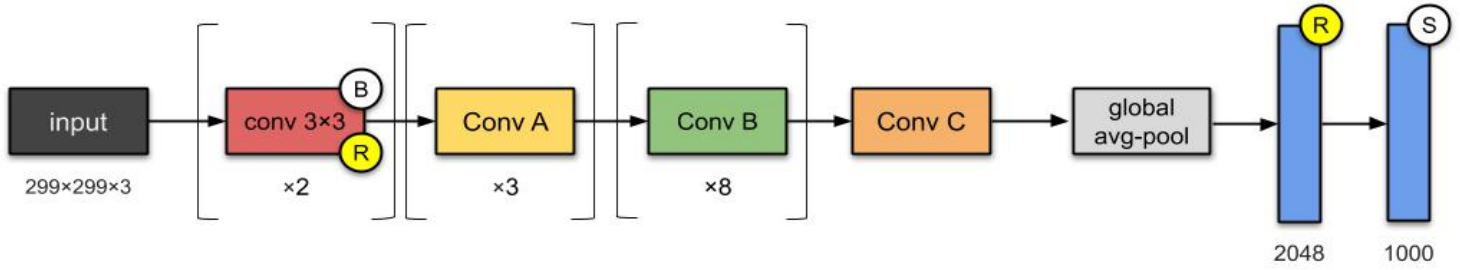


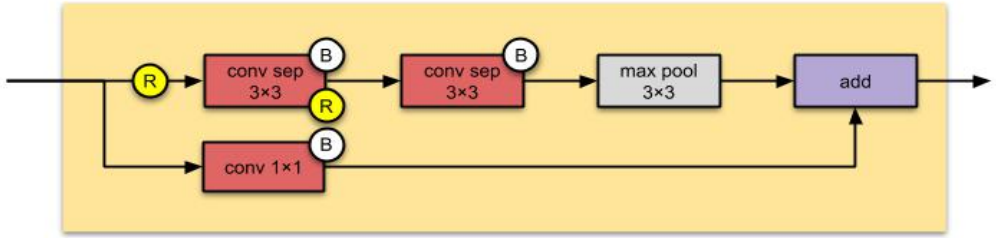
Conv block



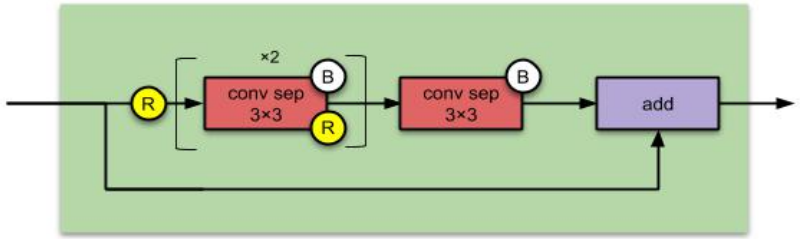
Identity block

CNN Variations - Xception (2016)

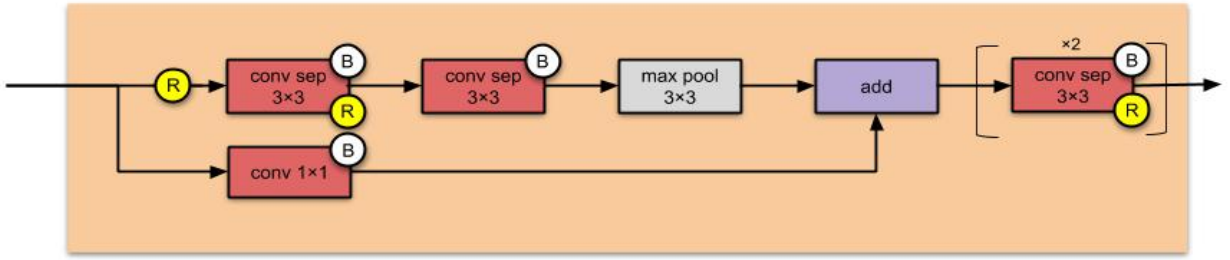




Conv A

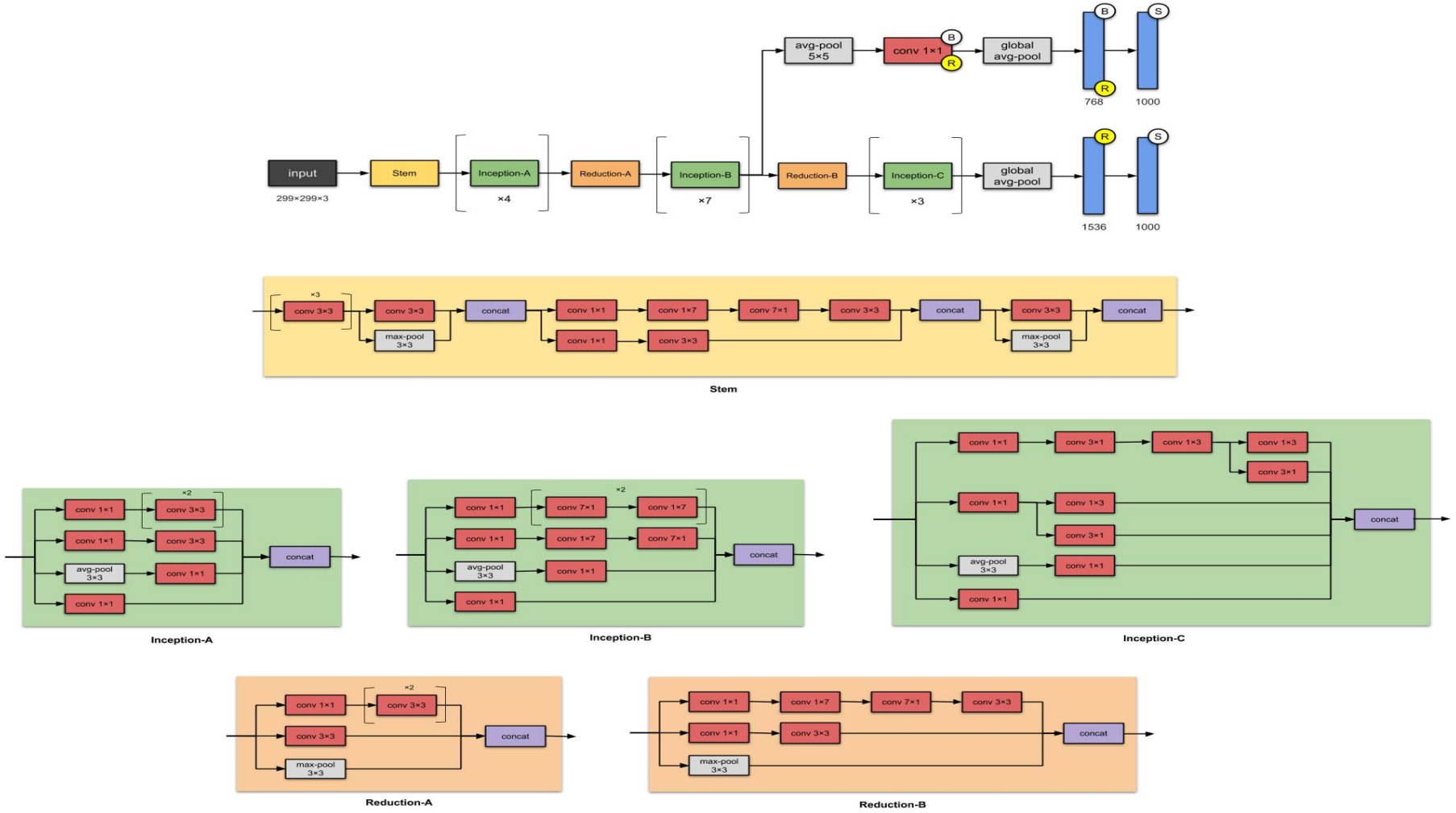


Conv B

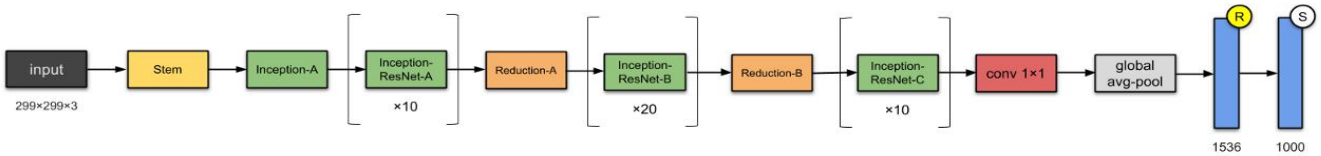


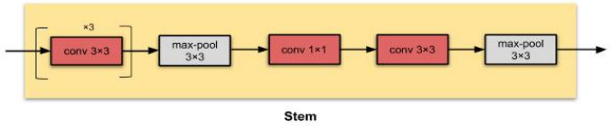
Conv C

CNN Variations - Inception-v4 (2016)

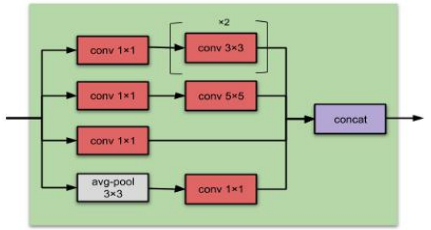


CNN Variations - Inception-ResNet-V2 (2016)

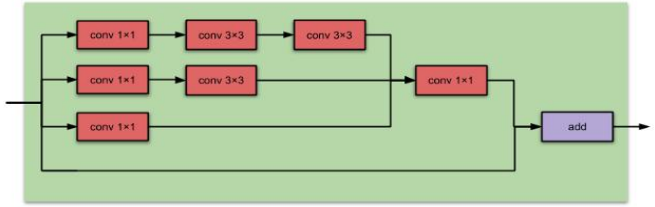




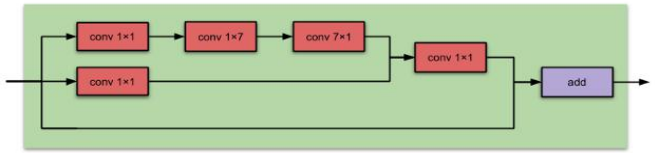
Stem



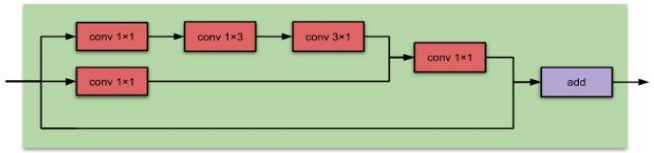
Inception-A



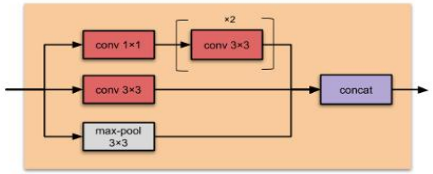
Inception-ResNet-A



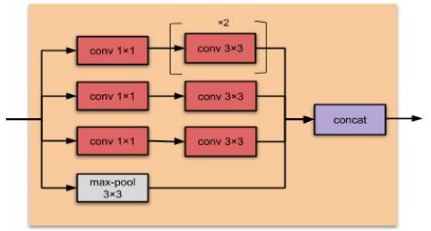
Inception-ResNet-B



Inception-ResNet-C

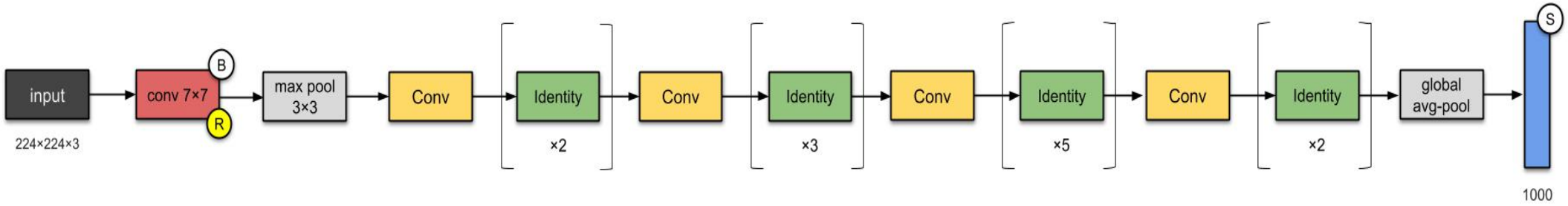


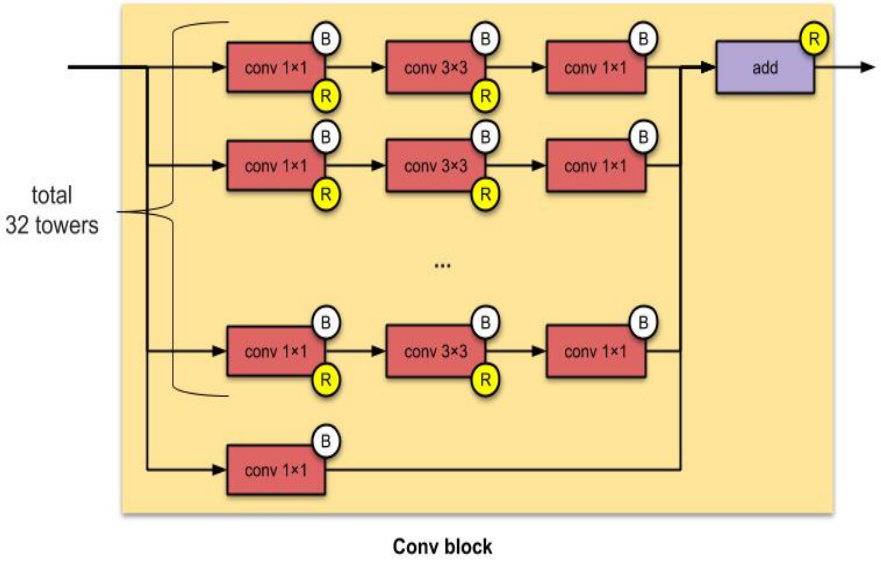
Reduction-A



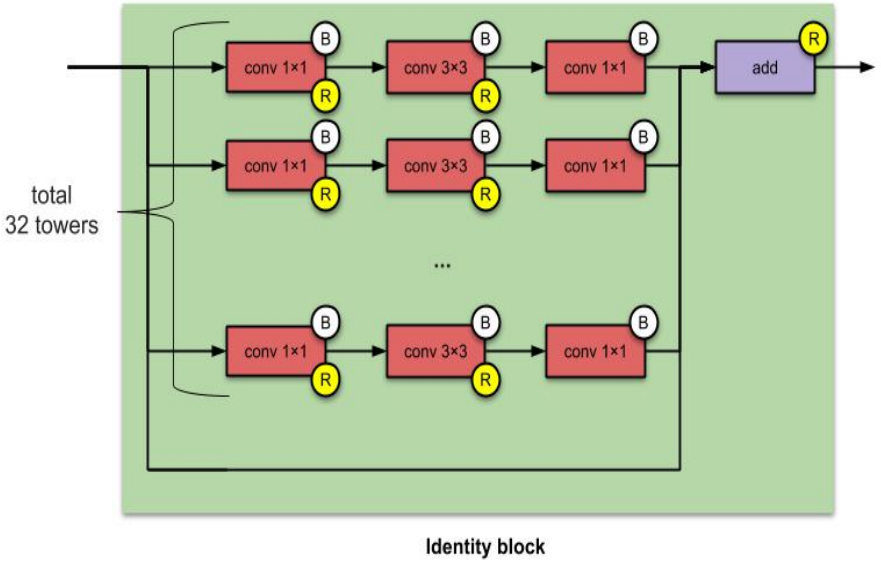
Reduction-B

CNN Variations - ResNeXt-50 (2017)





Conv block



Identity block

Using Pretrained Models from Keras

In general, you won't have to implement standard models like GoogLeNet or ResNet manually, since pretrained networks are **readily available** with a single line of code in the **keras.applications** package.

For example, you can load the **ResNet-50 model**, pretrained on **ImageNet**, with the following line of code:

```
model = keras.applications.resnet50. ResNet50
                                    (weights= "imagenet")
```

Using Pretrained Models from Keras

Available models :

Models for image classification with weights trained on ImageNet:

1) Xception
2) VGG16
3) VGG19
4) ResNet, ResNetV2
5) InceptionV3
6) InceptionResNetV2
7) MobileNet
8) MobileNetV2
9) DenseNet
10) NASNet

https://keras.io/applications/

Using Pretrained Models from Keras

Documentation for individual models

Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Xception	88 MB	0.790	0.945	22,910,480	126
VGG16	528 MB	0.713	0.901	138,357,544	23
VGG19	549 MB	0.713	0.900	143,667,240	26
ResNet50	98 MB	0.749	0.921	25,636,712	-
ResNet101	171 MB	0.764	0.928	44,707,176	-
ResNet152	232 MB	0.766	0.931	60,419,944	-
ResNet50V2	98 MB	0.760	0.930	25,613,800	-
ResNet101V2	171 MB	0.772	0.938	44,675,560	-
ResNet152V2	232 MB	0.780	0.942	60,380,648	-
InceptionV3	92 MB	0.779	0.937	23,851,784	159
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
MobileNet	16 MB	0.704	0.895	4,253,864	88
MobileNetV2	14 MB	0.713	0.901	3,538,984	88
DenseNet121	33 MB	0.750	0.923	8,062,504	121
DenseNet169	57 MB	0.762	0.932	14,307,880	169
DenseNet201	80 MB	0.773	0.936	20,242,984	201
NASNetMobile	23 MB	0.744	0.919	5,326,716	-
NASNetLarge	343 MB	0.825	0.960	88,949,818	-

The top-1 and top-5 accuracy refers to the model's performance on the ImageNet validation dataset.

Using Pretrained Models from Keras

```
model = keras.applications.resnet50.ResNet50(weights="imagenet")

# you first need to ensure that the images have the right size
# ResNet-50(224 × 224)
images_resized = tf.image.resize(images, [224, 224])

# Each model provides a preprocess_input() function
inputs = keras.applications.resnet50.preprocess_input(images_resized * 255)

# Now we can use the pretrained model to make predictions
Y_proba = model.predict(inputs)
```

Using Pretrained Models from Keras

```
import tensorflow as tf
from tensorflow import keras
model = keras.applications.vgg16.VGG16(weights=None)

# The model's summary() method displays all the model's layers
print(model.summary())
```

include_top: whether to include the top layers of the network or not (**False, True**).
weights: one of **None** (random initialization) or **'imagenet'** (pre-training on ImageNet).

Pretrained Models for Transfer Learning

- If you want to build an image classifier but you do not have **enough training data**, then it is often a good idea to **reuse the lower layers of a pretrained model**.

- **For example Xception model**, we exclude the top of the network by setting **include_top=False**: this excludes the global average pooling layer and the dense output layer. We then add our own layers. Finally, we create the Keras Model:

Pretrained Models for Transfer Learning

```
base_model = keras.applications.xception.Xception(weights="imagenet",include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
output = keras.layers.Dense(n_classes, activation="softmax")(avg)
model = keras.Model(inputs=base_model.input, outputs=output)

for layer in base_model.layers:
    layer.trainable = False

optimizer = keras.optimizers.SGD(lr=0.2, momentum=0.9, decay=0.01)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer, metrics=["accuracy"])
history = model.fit(train_set, epochs=5, validation_data=valid_set)

for layer in base_model.layers:
    layer.trainable = True

optimizer = keras.optimizers.SGD(lr=0.01, momentum=0.9, decay=0.001)
model.compile(...)
history = model.fit(...)
```

Thank you for your attention

link of the code:

https://github.com/hichemfelouat/my-codes-of-machine-learning/blob/master/Transfer_learning.py