ML in Applications

Dipartimento di Automatica e Informatica
Politecnico di Torino, Torino, ITALY

Lab 4

Tensorflow - Introduction

Tensors and variables

- TensorFlow operates on multidimensional arrays called **tensors** objects
- On tensors tf implements:
  - Canonical match operations
  - ML specialized ops
- Tensors are **immutable**
- To store model weights TensorFlow use a **tf.Variable**

```
[11]
import tensorflow as tf

x = tf.constant([[1., 2., 3.],
                 [4., 5., 6.]])

print(x)
print(x.shape)
print(x.dtype)
```

```
tf.Tensor(
[[1. 2. 3.]
 [4. 5. 6.]], shape=(2, 3), dtype=float32)
(2, 3)
<dtype: 'float32'>
```

```
[12]
x + x
```

```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[ 2.,  4.,  6.],
       [ 8., 10., 12.]], dtype=float32)>
```

```
[13]
tf.nn.softmax(x, axis=-1)
```

```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[0.09003057, 0.24472848, 0.66524094],
       [0.09003057, 0.24472848, 0.66524094]], dtype=float32)>
```

Automatic differentiation

- Optimization through **gradient descent** is a ML milestone
- To compute gradients tf implements automatic differentiation

```
[15]
x = tf.Variable(1.0)

def f(x):
  y = x**2 + 2*x - 5
  return y

f(x)
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=-2.0>
```

```
[16]
with tf.GradientTape() as tape:
  y = f(x)

g_x = tape.gradient(y, x)  # g(x) = dy/dx

g_x
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=4.0>
```

The *tf.GradientTape* API computes the gradient with respect to some inputs

Automatic differentiation

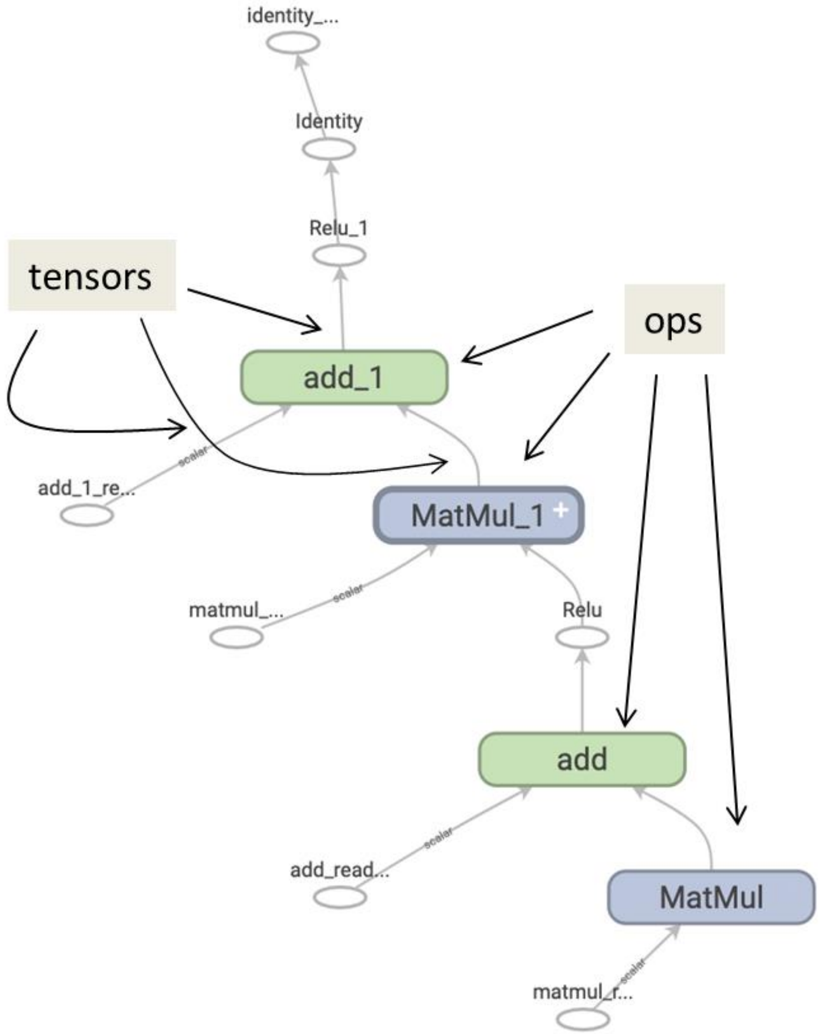
- **Gradient** of a loss **with respect to two variables**

```
[3]
import tensorflow as tf

w = tf.Variable(tf.random.normal((3, 2)), name='w')
b = tf.Variable(tf.zeros(2, dtype=tf.float32), name='b')
x = [[1., 2., 3.]]

with tf.GradientTape(persistent=True) as tape:
  y = x @ w + b
  loss = tf.reduce_mean(y**2)

[dl_dw, dl_db] = tape.gradient(loss, [w, b])
print(dl_dw)
print(dl_db)
```

```
tf.Tensor(
[[0.07654426 0.67659944]
 [0.15308851 1.3531989 ]
 [0.22963277 2.0297983 ]], shape=(3, 2), dtype=float32)
tf.Tensor([0.07654426 0.67659944], shape=(2,), dtype=float32)
```

Graphs

- In the previous cases tf was run **eagerly**
- Graph execution benefits:
  - Portability outside python (mobile applications, embedded devices, …)
  - Efficient execution
- Data structures that contain:
  - a set of **units of computation** (*tf.Operation)*
  - **units of data** that flow between operations (*tf.Tensor*)
- Graph can be saved, run, and restored all without the original Python code (and withput Python interpreter)
- Graphs are also optimized, allowing the compiler to:
  - Separate sub-parts of a computation that are independent and split them between threads or devices
  - Simplify arithmetic operations by eliminating common subexpressions
  - […]



Graphs

- In short, graphs let your tf **run fast**, run **in parallel**, and run **efficiently on multiple devices**
- We can still define our machine learning models in Python, and then **automatically construct graphs when we need** them
- *tf.function* take a canonical function as input and returns *Function,* a Python callable that build tf graphs from the python function
- On the outside, a *Function* looks like a regular function, but it encapsulates several *tf.Graphs* behind one API
- tf.function **applies** to a function and **all other functions it calls**

```
def inner_function(x, y, b):
  x = tf.matmul(x, y)
  x = x + b
  return x

# Use the decorator to make `outer_function` a `Function`.
@tf.function
def outer_function(x):
  y = tf.constant([[2.0], [3.0]])
  b = tf.constant(4.0)

  return inner_function(x, y, b)

# Note that the callable will create a graph that
# includes `inner_function` as well as `outer_function`.
outer_function(tf.constant([[1.0, 2.0]])).numpy()
```

Modules, layers, and models

- To do ML in TensorFlow, you are likely to need to define, save, and restore a model, which abstractly is
  - A function that computes something on tensors (a forward pass)
  - Some variables that can be updated in response to training
- Most models are made of layers
- **Layers** are **functions with a known mathematical structure** that can be reused and have **trainable variables**
- We will use **Keras**, high-level implementations of layers and models, built on the same foundational class *tf.Module*

Keras

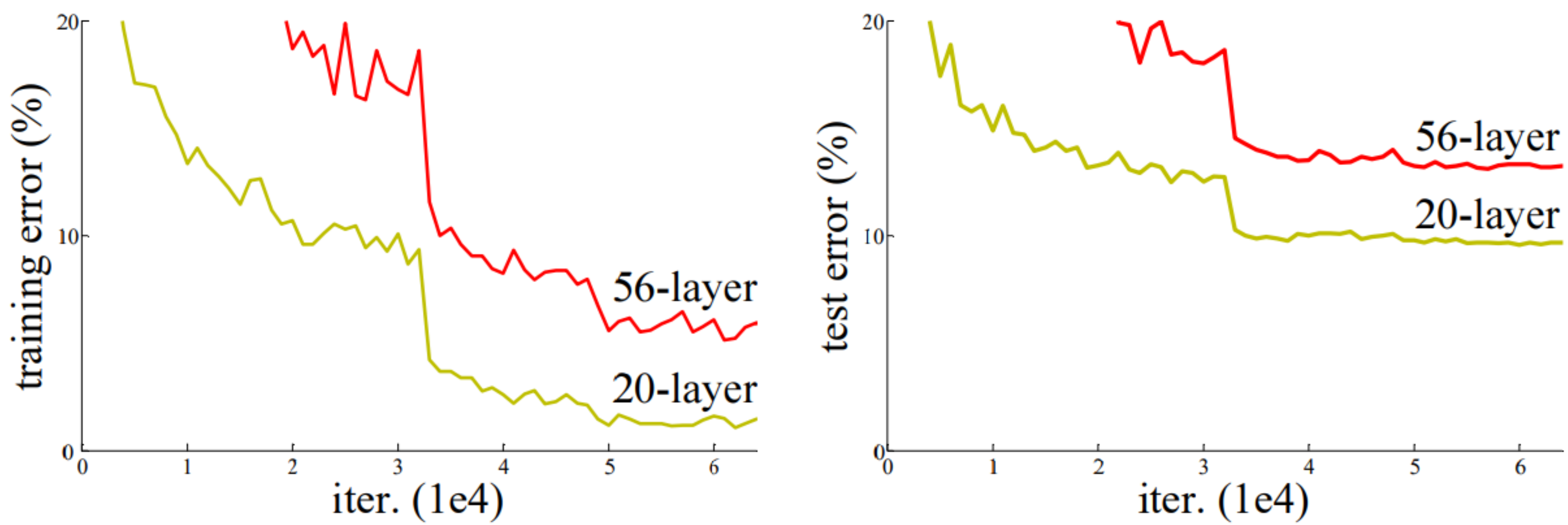
- ***tf.keras.layers.Layer*** is the base class of all Keras layers, and it inherits from *tf.Module*
- Also provides a full-featured model class called ***tf.keras.Model***
- In general, Layer class is used to define inner computation blocks, and the Model class to define the outer model
- The Model class has the same API as Layer but exposing also:
  - Built-in training, evaluation, and prediction loops
  - The list of its inner layers, via the model.layers property.
  - Saving and serialization APIs
- See Making new Layers and Models via subclassing

Assignment

ResNet

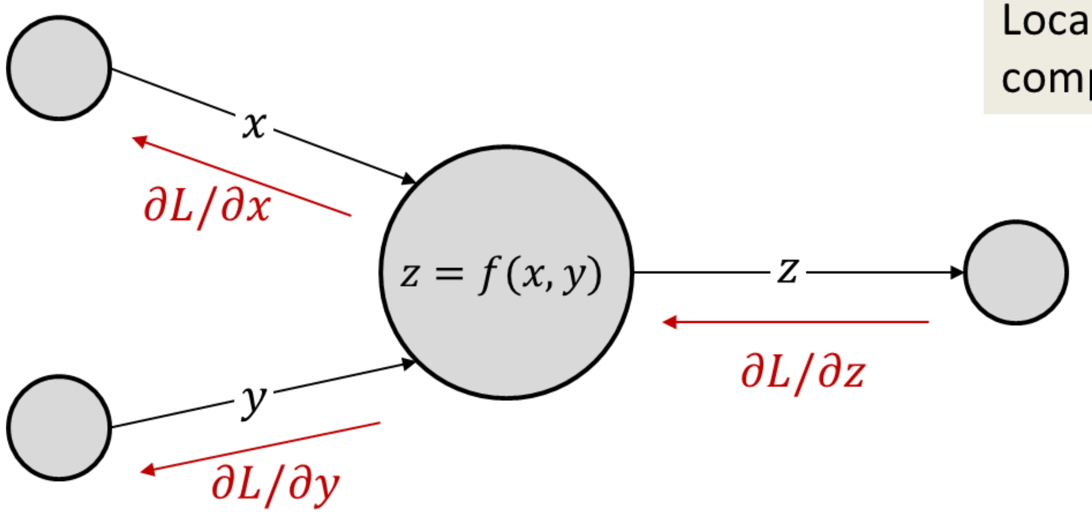
ResNet - Intuition

- Experimentally, with the network **depth increasing**, **accuracy** gets saturated (which might be unsurprising) and then **degrades rapidly**
- Unintuitively more layers = less accuracy
- This is due to two reasons:
  - The vanishing of the gradient
  - The degradation problem



ResNet – Gradient vanishing

- Traditional activation functions (e.g., the hyperbolic tangent) have gradients in (0,1]
- **Backpropagation computes gradients by the chain rule**
- **Multiplying** n of these **small numbers** will cause the gradient to decrease exponentially
- As the gradient keeps flowing backward to the initial layers, this value keeps getting multiplied by each local gradient
- The vanishing of the gradient prevents the learning of the early layers



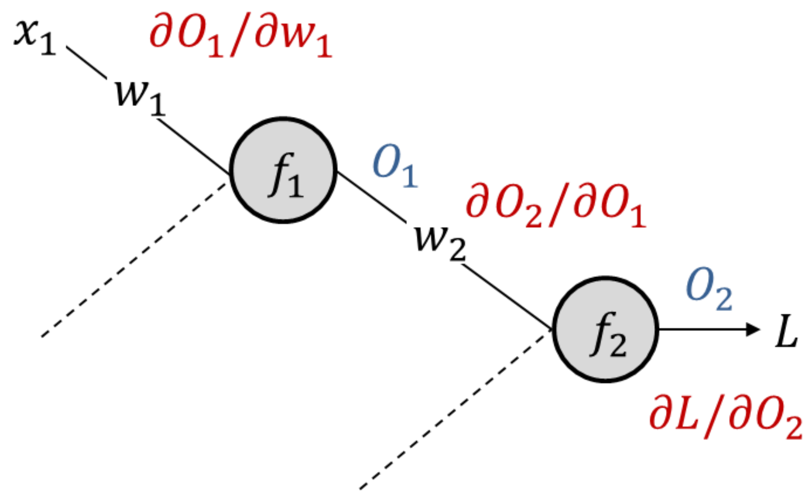
Local gradients of a neuron computed by backprop

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$$

$$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$$

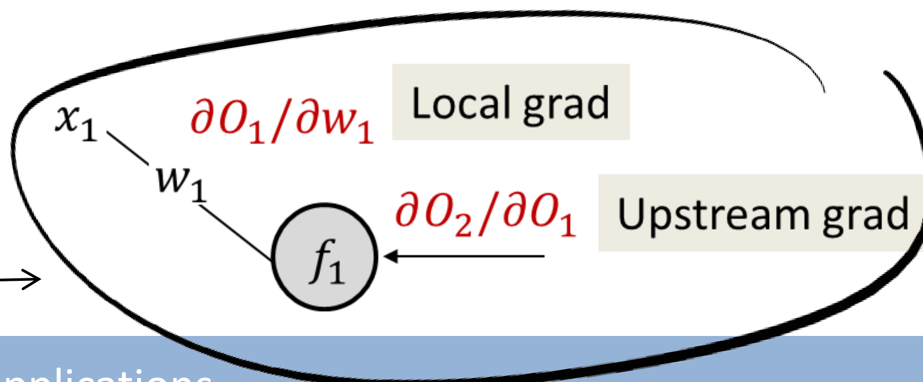
ResNet – Gradient vanishing

- A node in a computational graph can compute two things without even being aware of the rest of the graph :
  - the output of the node
  - the local gradient of the node.
- Local gradients of a node are the derivatives of the output of the node with respect to each of the inputs
- Essence of **backpropagation**: for a **single node**, we find the derivative of the output w.r.t a variable by **multiplying its local gradient with the upstream gradient** that we receive from the upstream step



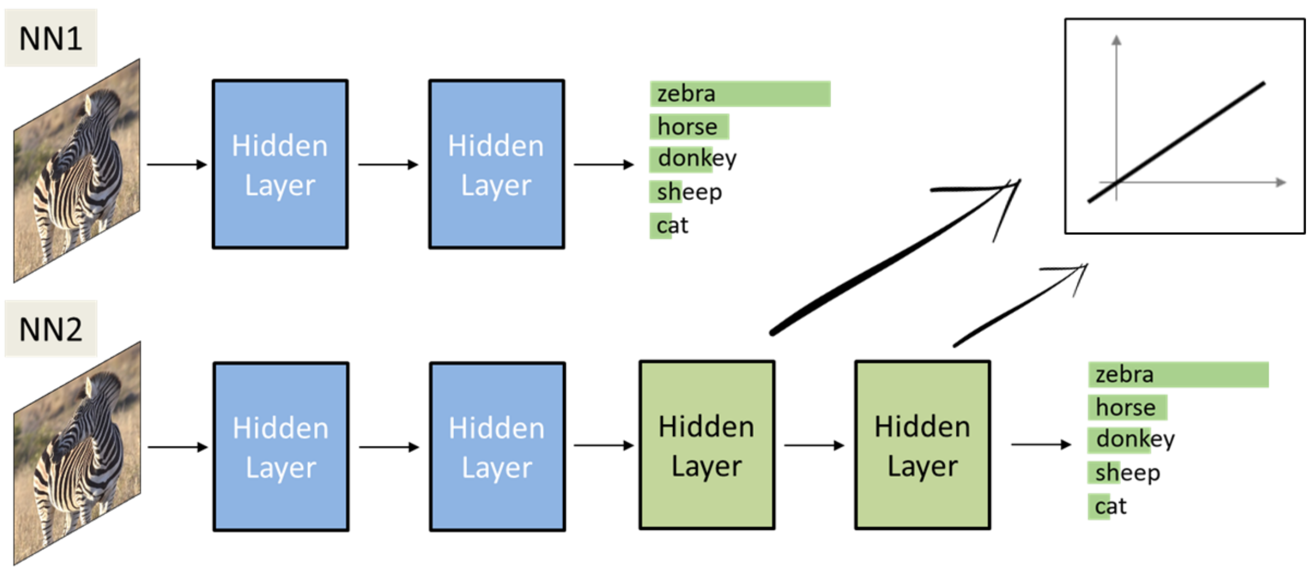
Chain rule

$$L(O_2(O_1(w_1))) \rightarrow \frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial O_2}\frac{\partial O_2}{\partial O_1}\frac{\partial O_1}{\partial w_1}$$



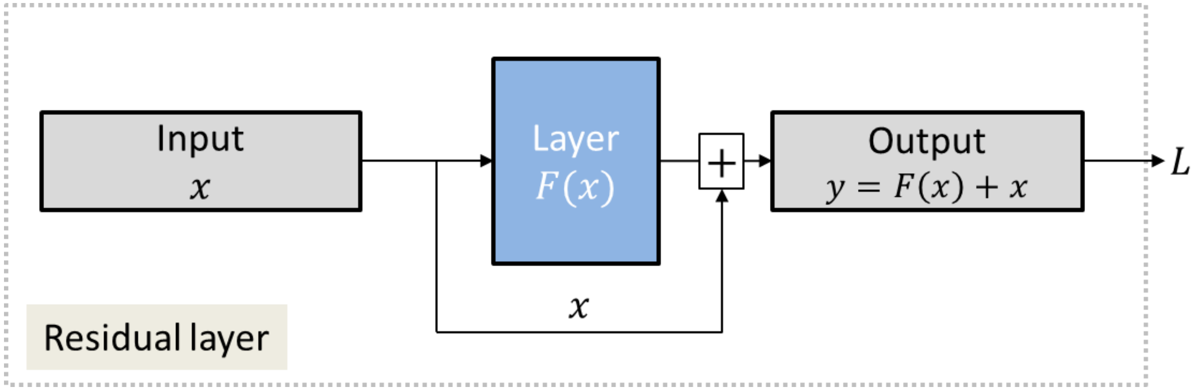
ResNet – Degradation problem

- **Theoretically**, the more layers you add to a neural network, the performance could either go up or stay the same, it should never go down
- Suppose NN1 in its *best possible state* (loss in global minimum)
- If we add more hidden layers (NN2), the new layers should learn the **identity function** $g(x) = x$ to preserve the current best state of the net
- Learning the identity function from scratch is extremely difficult: the huge number of weights and bias values must be modified to correspond to identity function



ResNet – Residual learning

- We introduce the *ResNet block*, a layer with **skip** connections

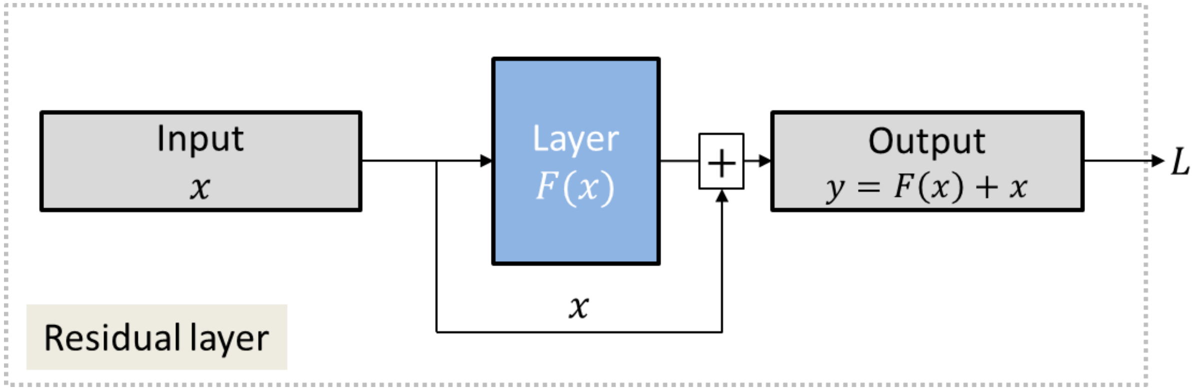


- Defining $y = F(x) + x$, the layer must learn $F(x) = y - x$, which is the **residual** mapping
- In this scenario, learning **the identity function** means learning the **zero** function $y = 0 + x = x$
- Easy to learn, set all weights to zero
- No vanishing gradient: $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial L}{\partial y}(F'(x) + 1) = \frac{\partial L}{\partial y} + \frac{\partial L}{\partial y}F'(x)$

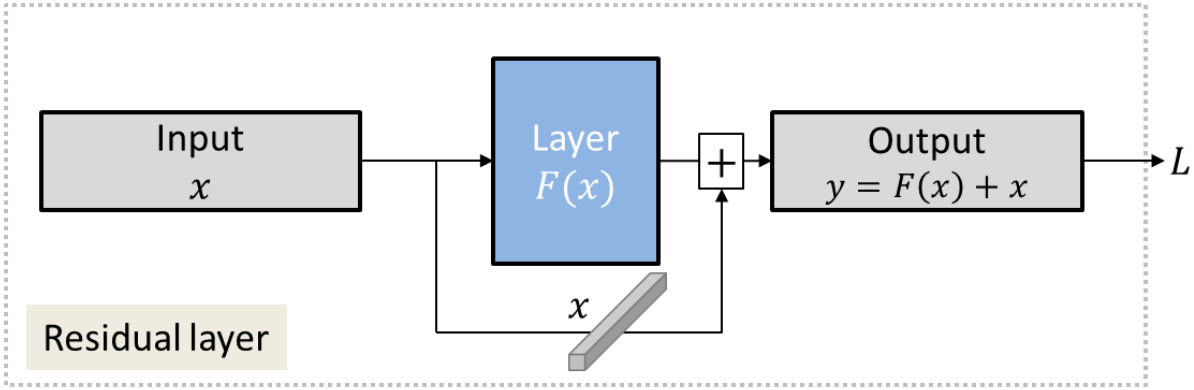
The error signal (incoming gradient) is passed through, unmodified by the local gradient

ResNet – Residual learning

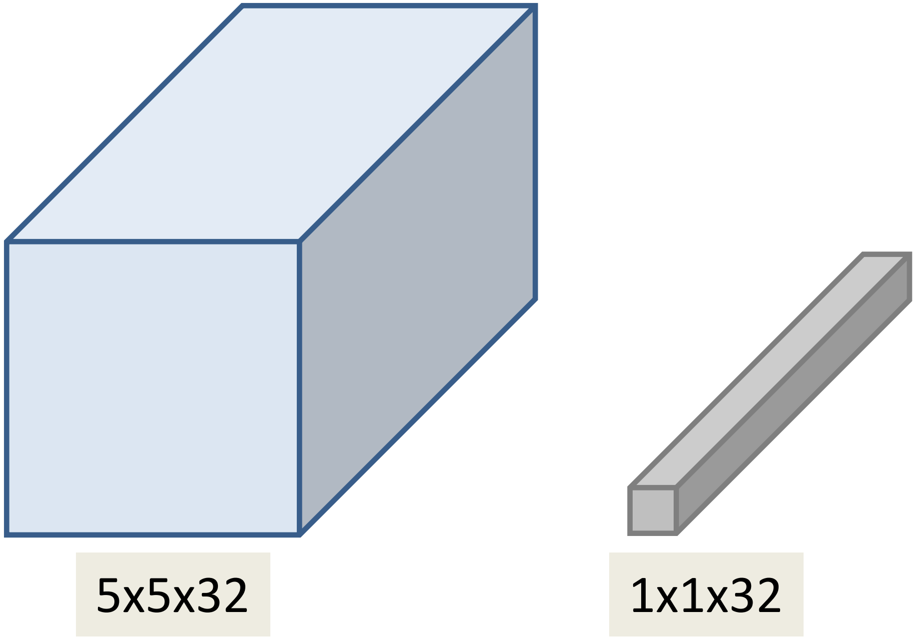
- The architecture depends on the dimension of the output w.r.t. the input



- If the input dimension has the same dimension as the output the block implements an identity function
- Otherwise in the skip connection we insert a **1x1 convolutional block**

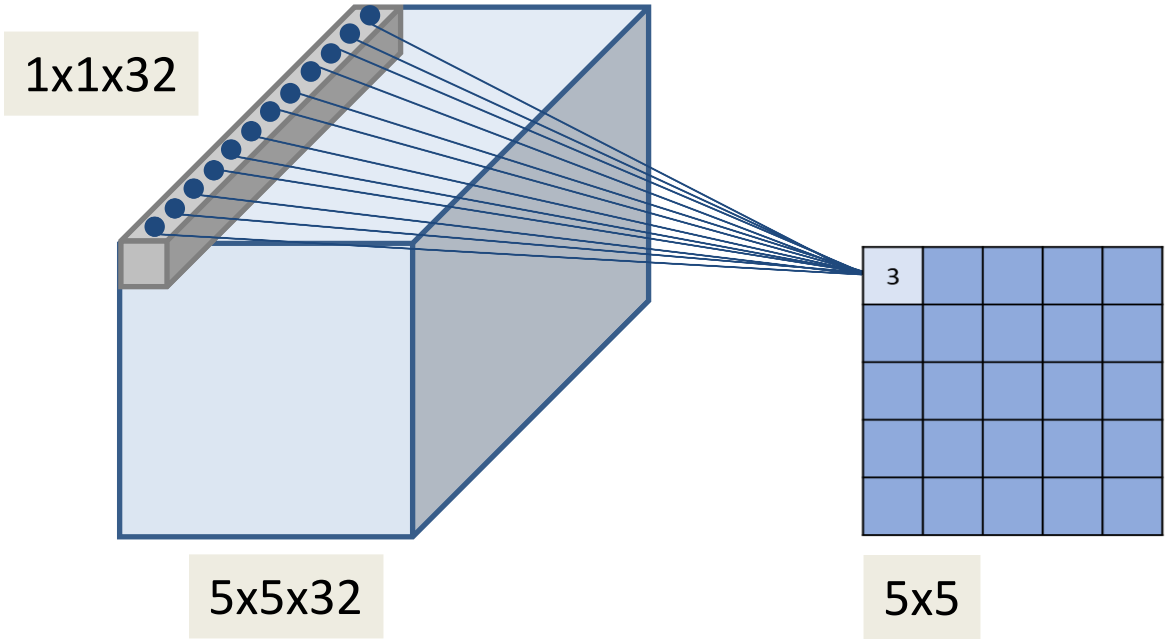


ResNet - 1x1 convolution



A 1x1 convolution will take an **element-wise product** and then apply a Relu non-linearity

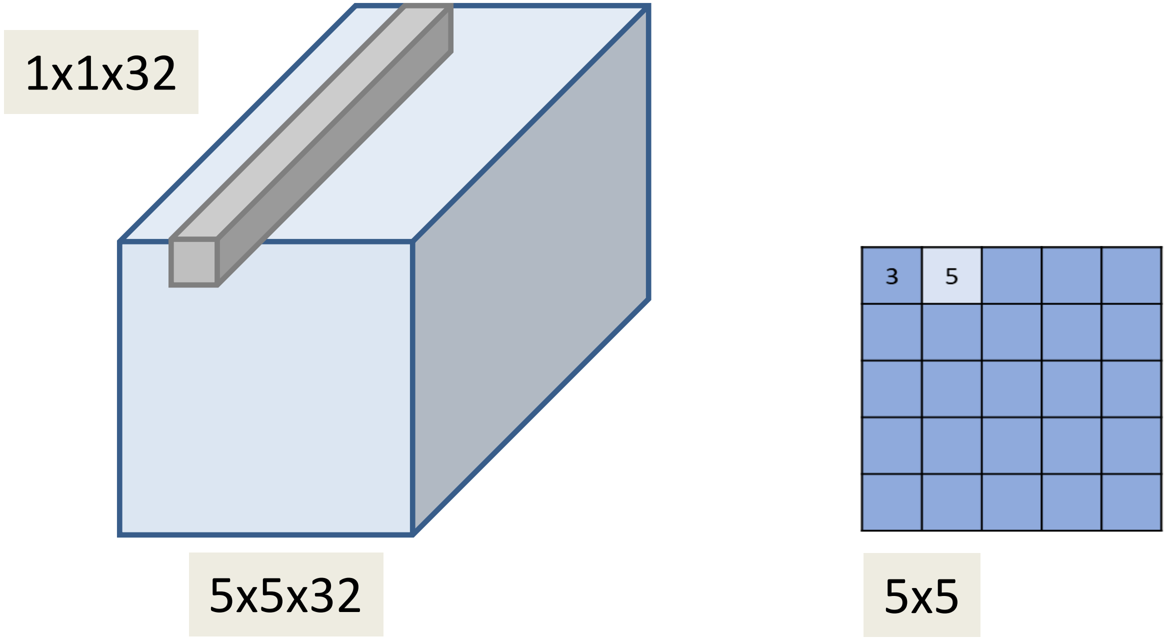
ResNet - 1x1 convolution



Can be see a neuron having as input 32 number multiplied by weights → Network In Network [1]

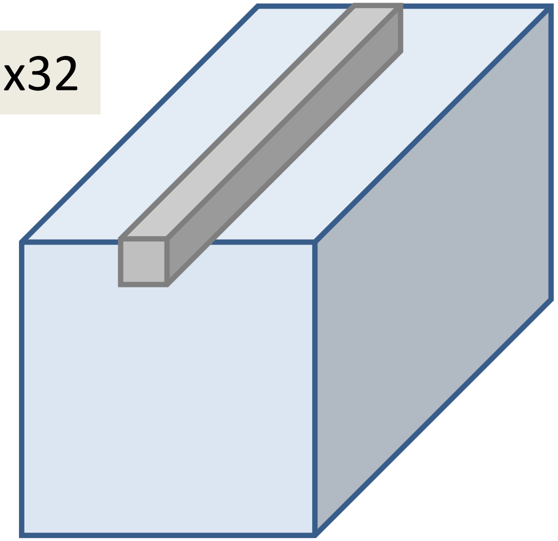
[1] Lin, Min, Qiang Chen, and Shuicheng Yan. "Network In Network." arXiv preprint arXiv:1312.4400 (2013).

ResNet - 1x1 convolution

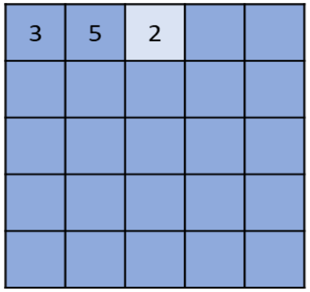


ResNet - 1x1 convolution

1x1x32

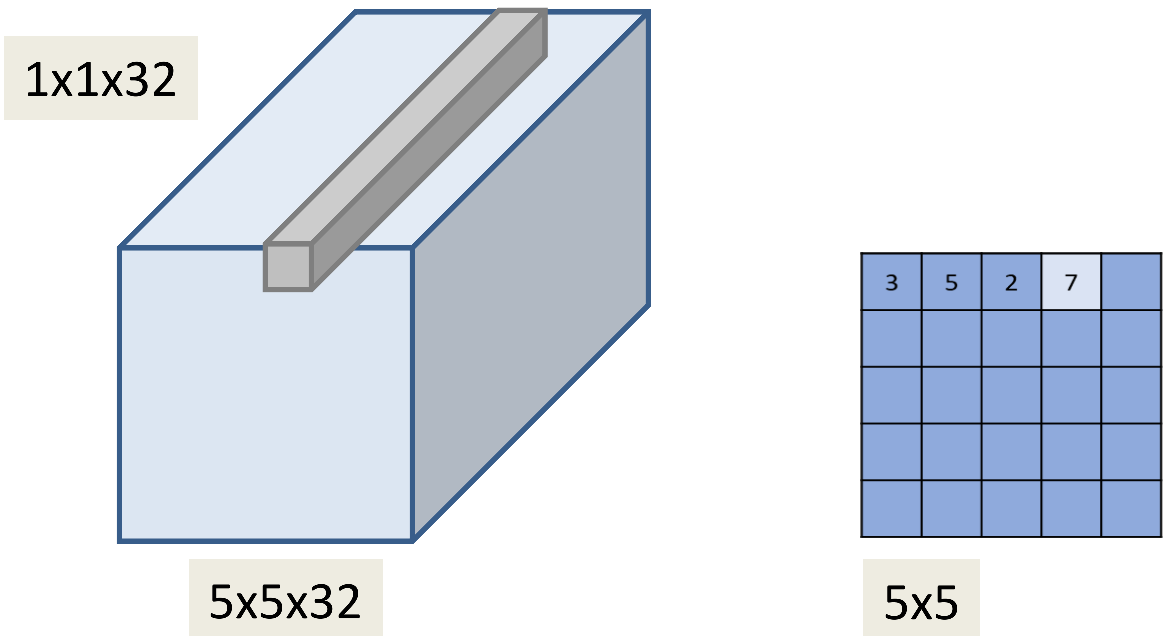


5x5x32

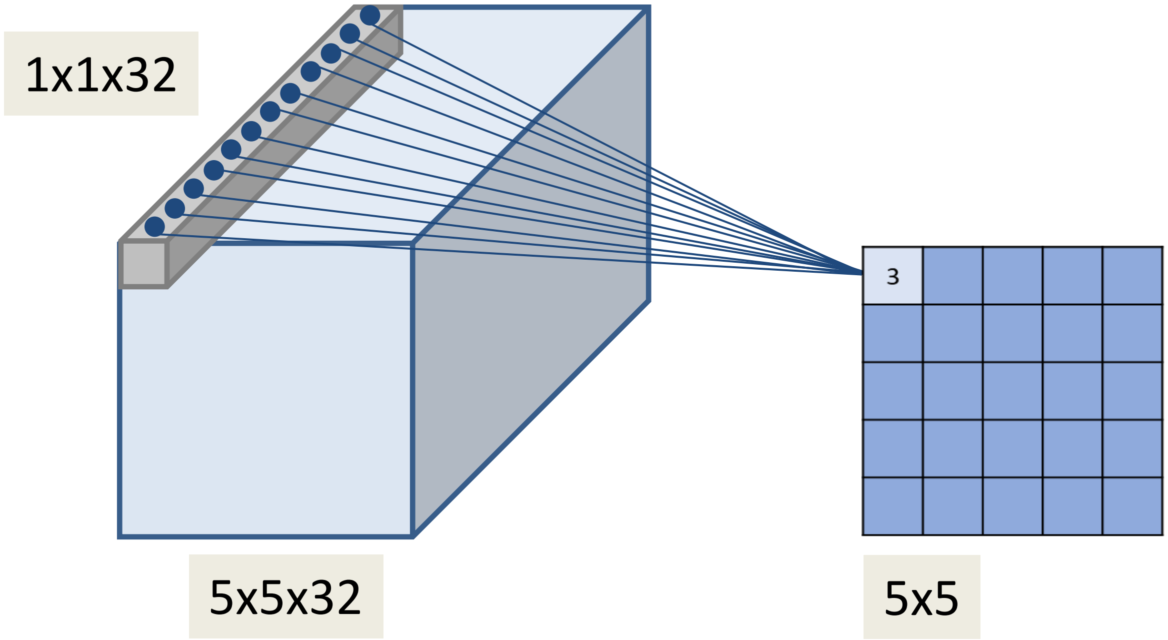


5x5

ResNet - 1x1 convolution

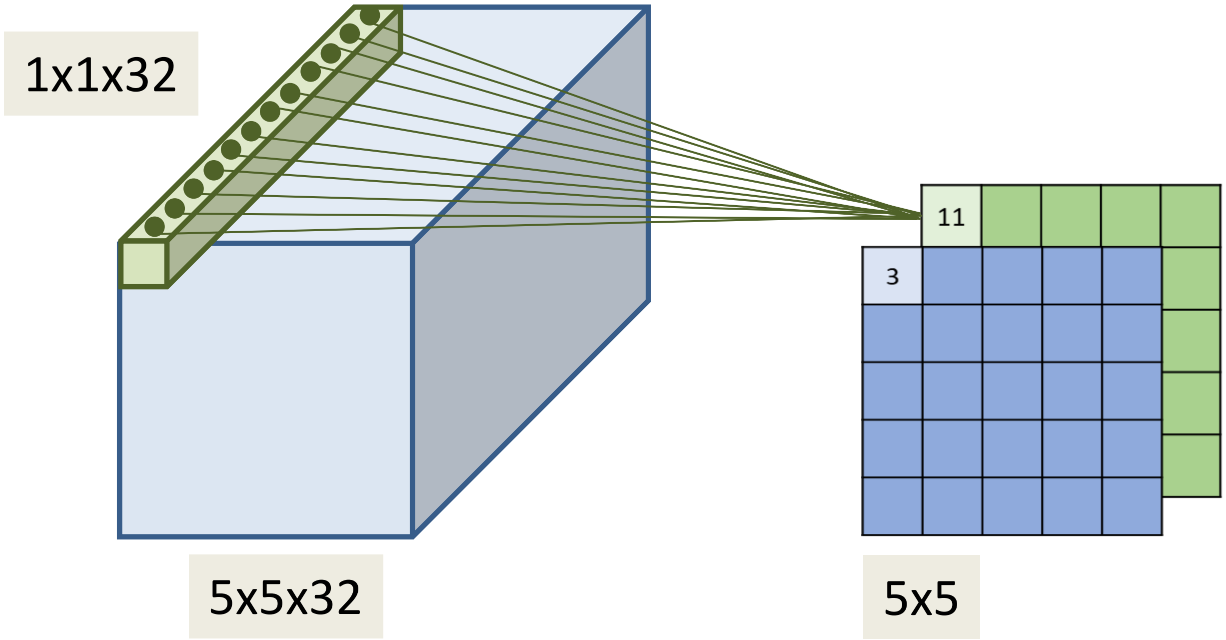


ResNet - 1x1 convolution



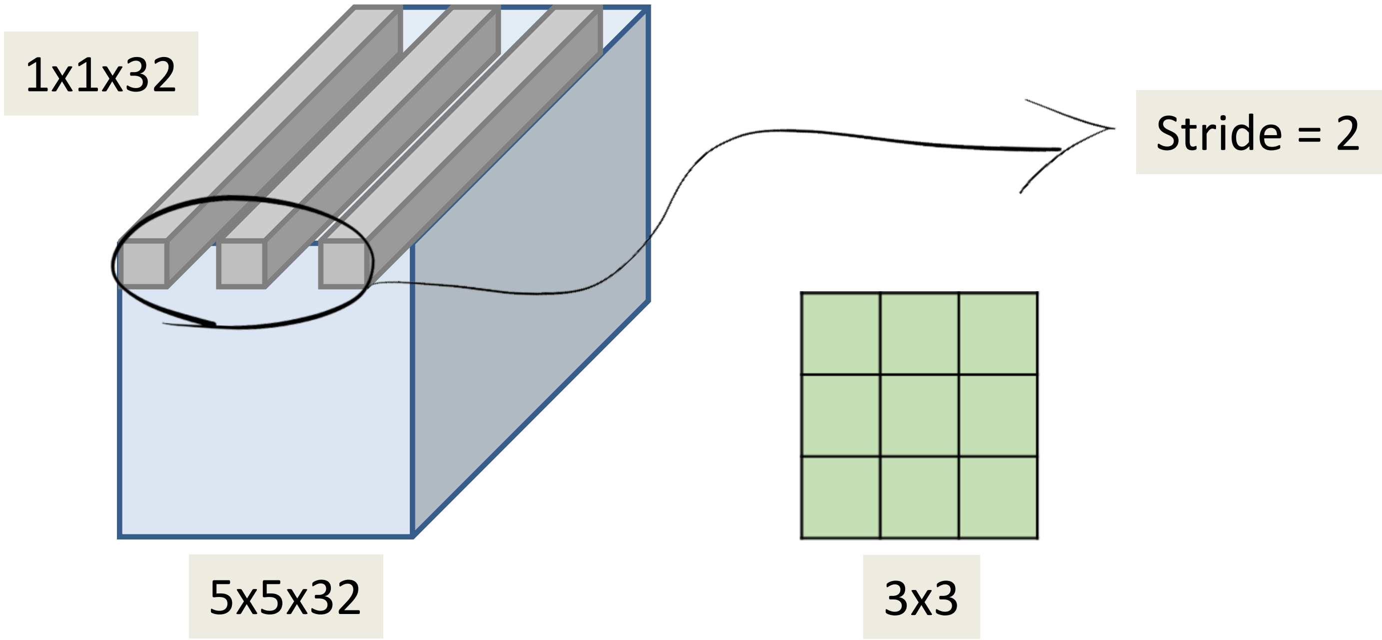
Using N different 1x1 convolutional filters we obtain volumes of different depth

ResNet - 1x1 convolution



Using N different 1x1 convolutional filters we obtain volumes of different depth

ResNet - 1x1 convolution



Concerning ResNet, since there's no pooling layer within the residual block, the dimension is downscaled by 1×1 convolution with strides 2