Introduction to Deep Learning

Lecture 6

Deep Learning Frameworks

Andrius Bernatavicius, 2021

Outline

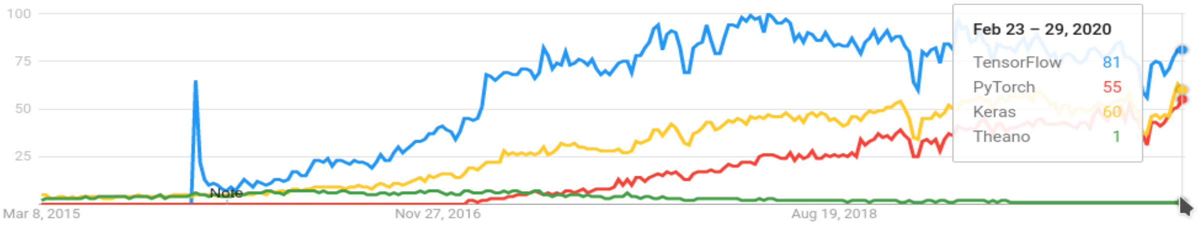
- **Introduction**
  - History of DL frameworks
- **Tensorflow**
  - Tensors, tensor operations
  - Computational graph
  - Keras integration
- **Practical examples**
  - Building networks in Keras
  - Tensorboard, callbacks

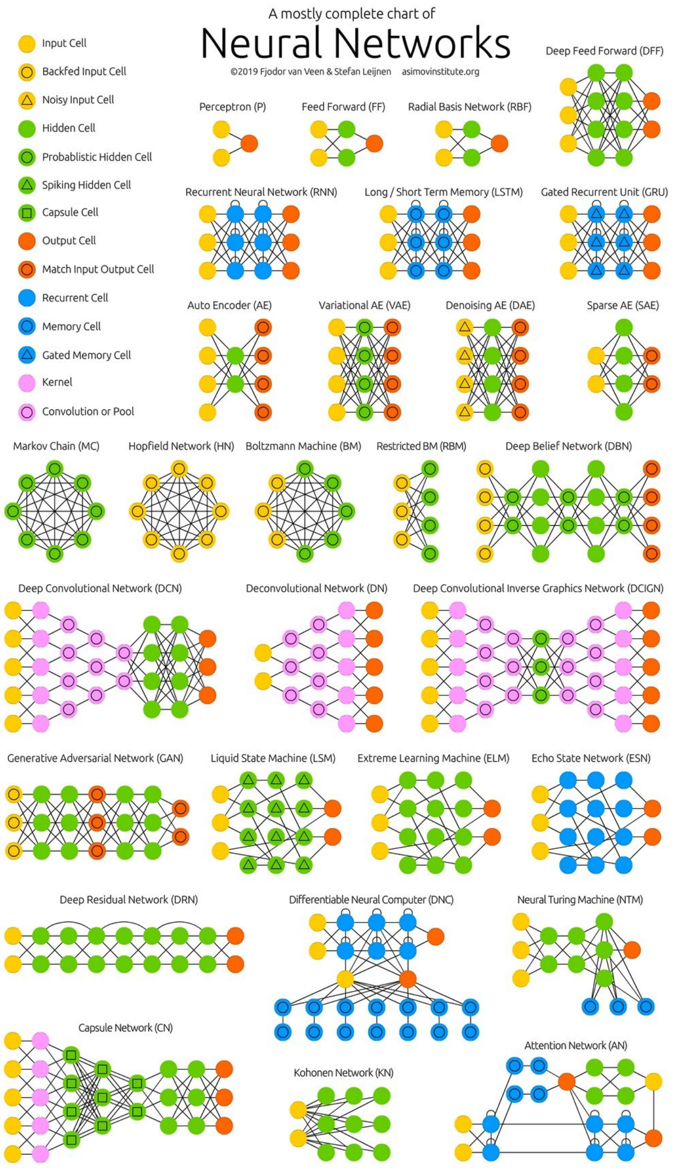
Deep Learning Frameworks

- Building neural networks from scratch is a time consuming process
- Many different libraries:
  - Theano (Montreal)
  - Caffe (Berkeley)
  - **Tensorflow** (Google)
  - **Torch** (Facebook)
  - CuDNN (Nvidia)
  - **Keras** (Theano / Tensorflow backend)
- Focus on high level programming language frontends for development
- Libraries increase model reproducibility, reduce errors, increase efficiency

Deep Learning Frameworks

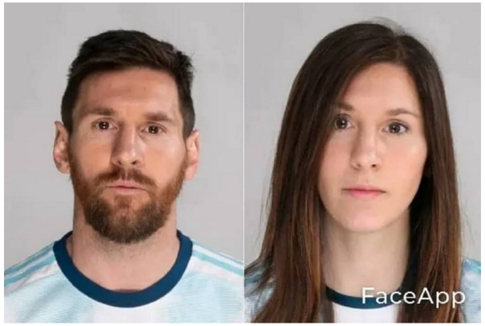
- Neural networks scale both with the amount of data and available computational resources
- Motivation to build efficient libraries that can scale and utilize various computational devices
- Main leaders in design – companies with top talent and a lot of data (Google, Facebook, Microsoft)
- Open source mindset – accelerated science, development





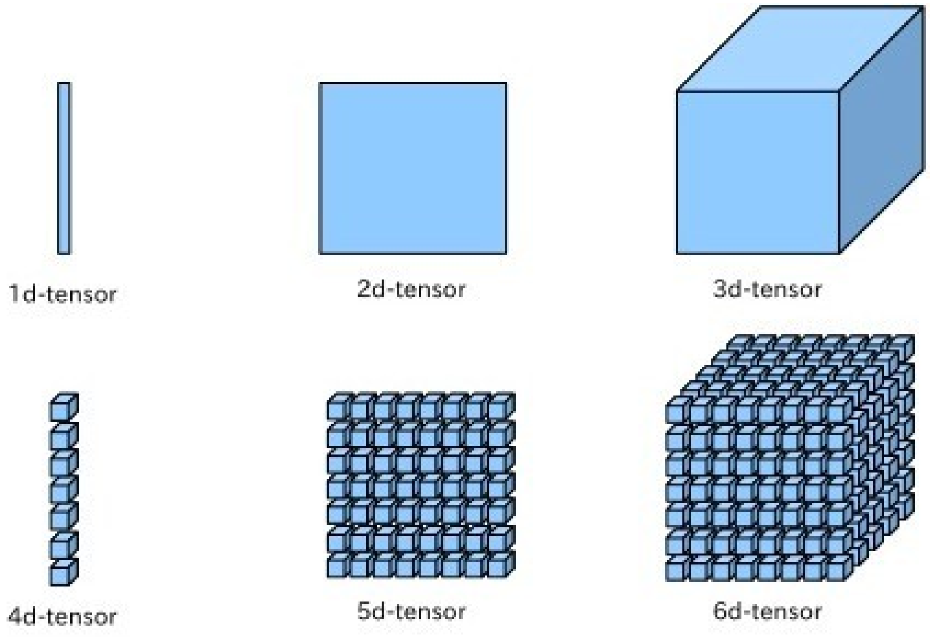
Use cases of TensorFlow / Torch

- A lot of Google products:
  - Search, Google translate, Gmail, image/voice recognition
- Self-driving car industry:
  - Tesla, CommaAI, Waymo, Uber
- Recommendations (ads, products):
  - Facebook, Google, Booking.com, etc.
- Mobile apps:
  - Anything relating to computer vision, generative models, natural language processing
- Dozens of other use cases and thousands of companies using neural network models
- In less than 10 years (since 2012) all these companies started focusing very heavily on **neural networks**



Tensors

- Generalization of an N-dimensional array
- Attributes:
  - Rank / degree / order / dimensionality
  - Shape
  - Datatype
- Operations:
  - Algebraic operations
  - Reshaping – increasing / decreasing rank
  - Applying functions
    - Element wise
    - By reducing dimensionality (e.g. mean over axis)
- Efficient in terms of storage, retrieval
- Inherently parallel
- Easier to conceptualize computations



Tensor operations

- Algebraic operations
- Tensor multiplication:
  - 2^{nd} rank

$$[N * K] \times [K * M] = [N * M]$$

  - 3^{rd} rank → 2^{nd} rank

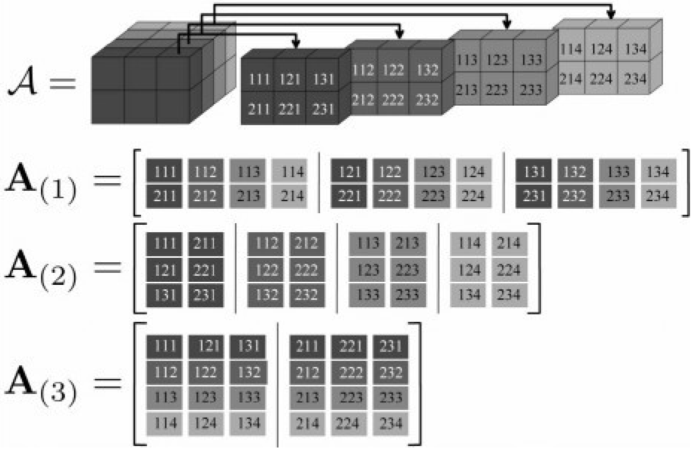
$$[N * K * L] \times [L * K * M] = [N * M]$$

- Applying functions
- Reshaping (Increasing/reducing rank)

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix}, \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix}$$

$$\mathbf{AB} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} & a_{11}b_{13} + a_{12}b_{23} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} & a_{21}b_{13} + a_{22}b_{23} \\ a_{31}b_{11} + a_{32}b_{21} & a_{31}b_{12} + a_{32}b_{22} & a_{31}b_{13} + a_{32}b_{23} \end{pmatrix}$$

$$\mathbf{BA} = \begin{pmatrix} b_{11}a_{11} + b_{12}a_{21} + b_{13}a_{31} & b_{11}a_{12} + b_{12}a_{22} + b_{13}a_{32} \\ b_{21}a_{11} + b_{22}a_{21} + b_{23}a_{31} & b_{21}a_{12} + b_{22}a_{22} + b_{23}a_{32} \end{pmatrix}$$

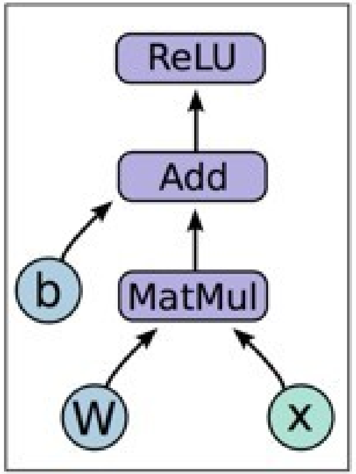


Perceptron layer example

- Weights of a fully connected layer - 2^{nd} rank Tensor

$$h = ReLU(Wx + b)$$

- **x** = $[N_{samples} * F_{features}]$ - *batch of input data*
- **W** = $[F_{features} * M_{nodes}]$ - *layer weights*
- **B** = $[\ 1 * M_{nodes}]$ - *layer biases*
- **Out** = $[N_{samples} * M_{nodes}]$
- *Example*: 100 samples of MNIST, 512 node perceptron layer
- [100 x 784] x [784 x 512] → Output = [100 x 512]
  Activations of each node for each data sample

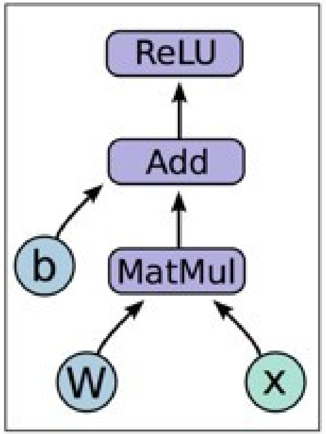


TensorFlow

- Optimized numerical computation/deep learning library
- Main idea: computational (data flow) graph
- Tensors, tensor operations, automatic differentiation
- Support for various computation devices, platforms (tensorflow lite)
- Python, Java, JS, Swift
- Tensorflow 2.0 – major overhaul

TensorFlow

$$h = ReLU(Wx + b)$$

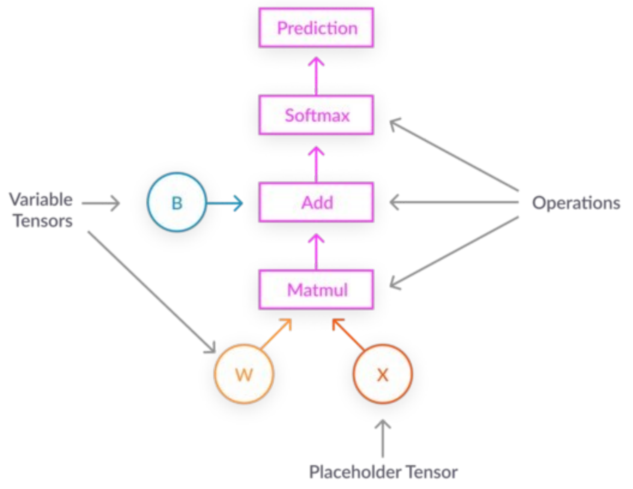


Numpy and TensorFlow analogies

- Both operate on [N-dimensional] arrays (tensors)
- More similarities:
  - Both are implemented in C/C++
  - Optimized routines for operations, targeted for specific hardware
  - Similar API, definitions for creating objects

TensorFlow (2)

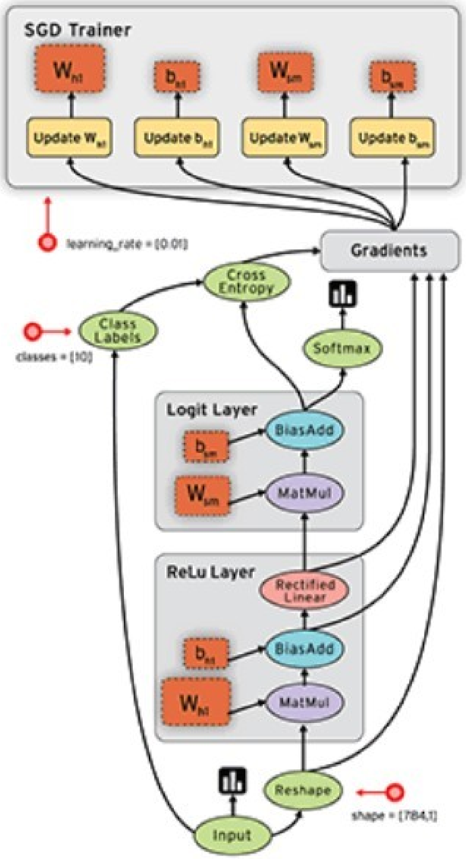
- Added functionality:
  - GPU, TPU support, scales to large distributed systems
  - Automatic differentiation of trainable model parameters
  - Tools for tracking training metrics (Tensorboard)
  - Various efficient implementations of neural network related mathematics (XLA compiler and CUDA/CUDNN integration)



COMPUTATIONAL GRAPH

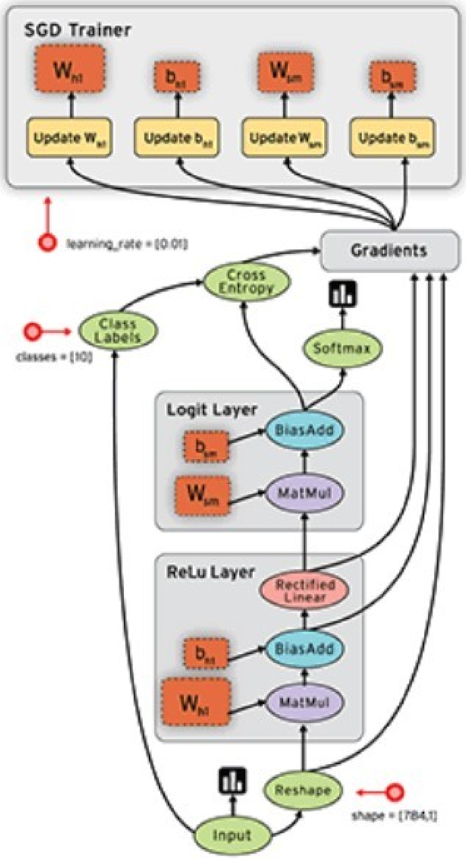
TensorFlow computational graph

- Two main components:
  - Tensors
    - Variables – trainable
    - Constants – e.g. data, hyperparameters
  - Tensor operations
    - Algebraic
    - Activation, loss functions
    - Gradient computation (derivatives, backpropagation)



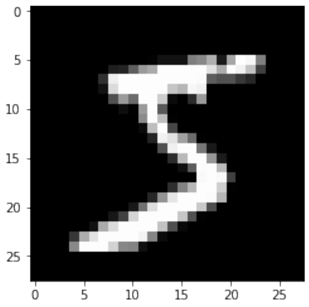
TensorFlow computational graph

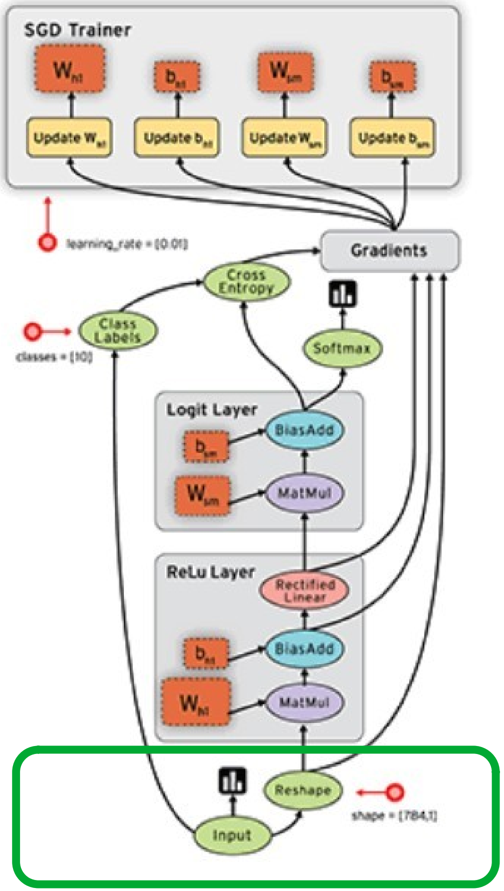
- High level components of a neural network computational graph:



TensorFlow computational graph

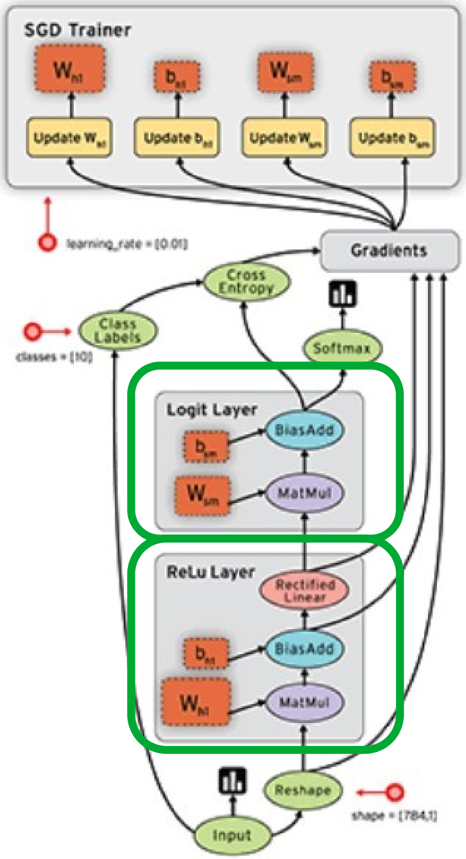
- High level components of a neural network computational graph:
  - Input data, preprocessing





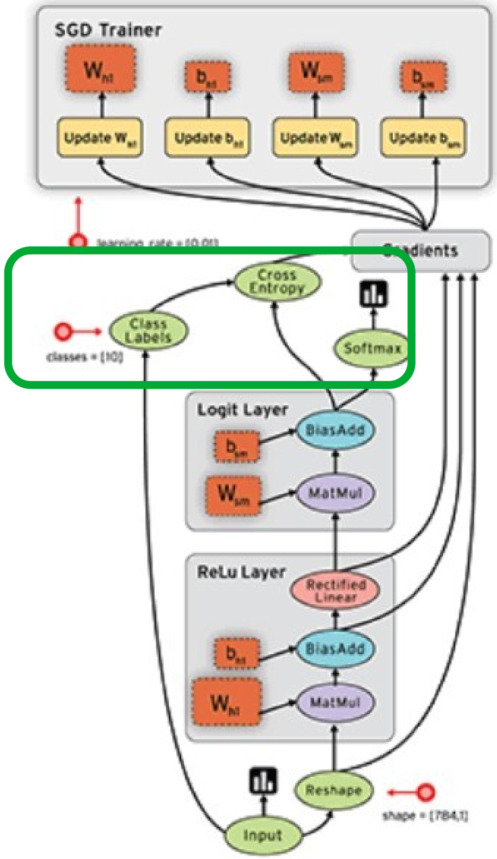
TensorFlow computational graph

- High level components of a neural network computational graph:
  - Input data, preprocessing
  - Layers



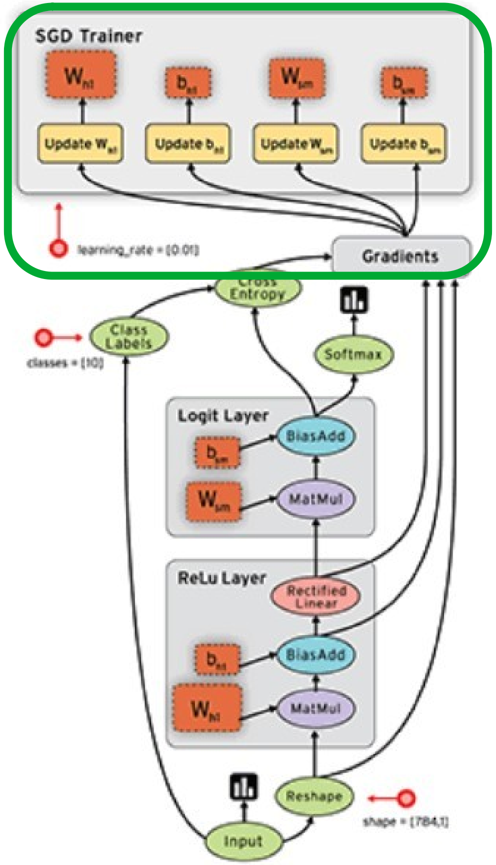
TensorFlow computational graph

- High level components of a neural network computational graph:
  - Input data, preprocessing
  - Layers
  - Loss function



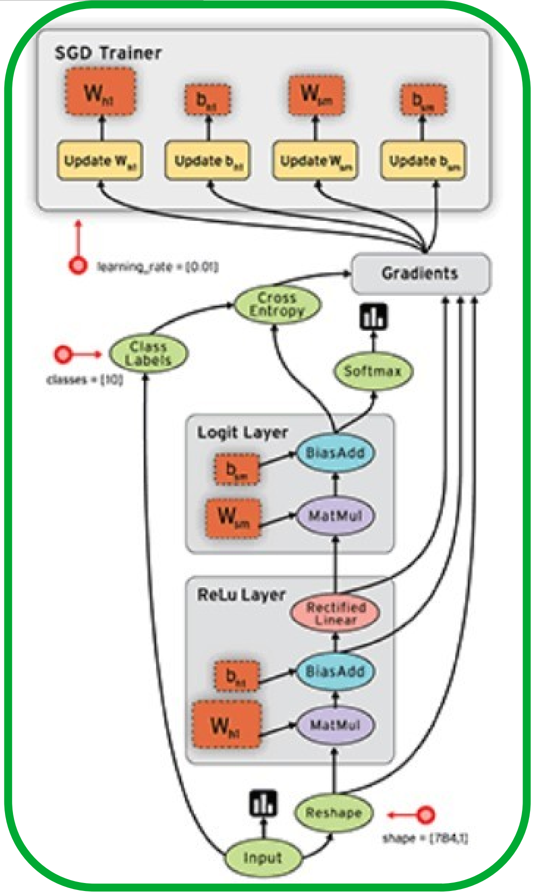
TensorFlow computational graph

- High level components of a neural network computational graph:
  - Input data, preprocessing
  - Layers
  - Loss function
  - Optimizer



TensorFlow computational graph

- High level components of a neural network computational graph:
  - Input data, preprocessing
  - Layers
  - Loss function
  - Optimizer
- Objective: *find optimal values of model parameters with respect to the loss function by backpropagating the error through the computational graph*



Keras

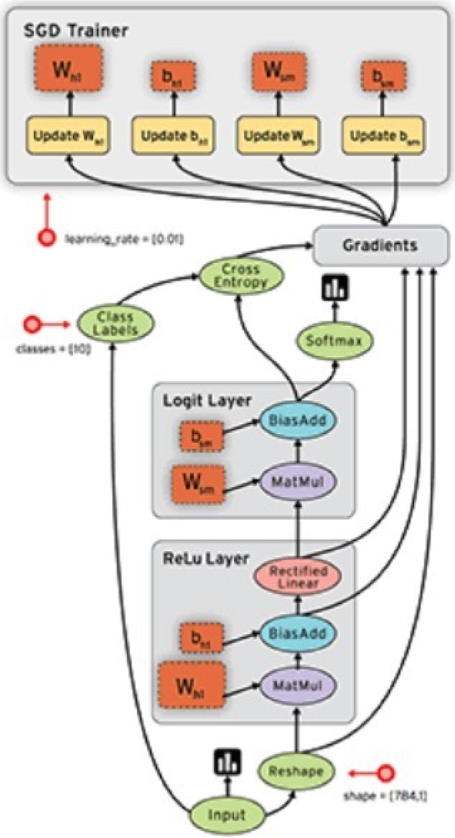
- Provides an accessible way of building high-performance deep learning models
- High level API for Theano and Tensorflow
- Part of Tensorflow 2.0
- Build models very quickly automate a lot of routines of building and tuning models.
- *Keras -> Tensorflow || Scikitlearn -> Numpy analogy*

Multilayer perceptron in Keras

- Single hidden layer with 512 nodes

```
import tensorflow as tf

model = tf.keras.Sequential()
model.add(tf.keras.layers.Input(shape=data.shape))
model.add(tf.keras.layers.Dense(512, activation='relu'))
model.add(tf.keras.layers.Dense(10, activation='softmax'))
optimizer = tf.keras.optimizers.SGD(lr=0.01)
loss = tf.keras.losses.categorical_crossentropy
model.compile(optimizer=optimizer, loss=loss)
```

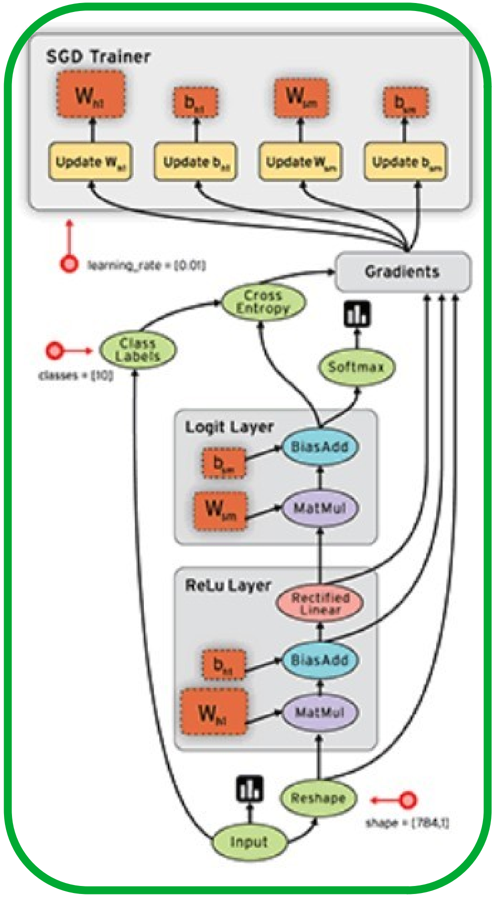


Multilayer perceptron in Keras

- Single hidden layer with 512 nodes

```
import tensorflow as tf

model = tf.keras.Sequential()
model.add(tf.keras.layers.Input(shape=data.shape))
model.add(tf.keras.layers.Dense(512, activation='relu'))
model.add(tf.keras.layers.Dense(10, activation='softmax'))
optimizer = tf.keras.optimizers.SGD(lr=0.01)
loss = tf.keras.losses.categorical_crossentropy
model.compile(optimizer=optimizer, loss=loss)
```

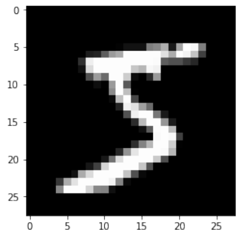


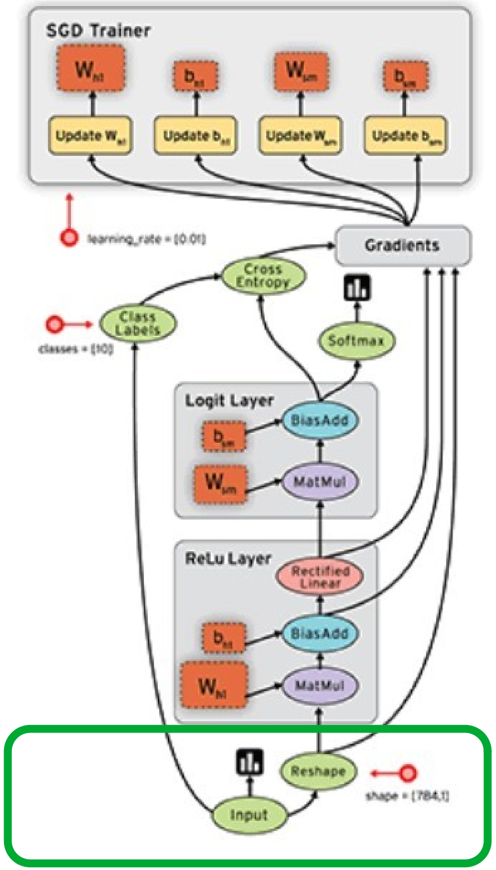
Multilayer perceptron in Keras

- Single hidden layer with 512 nodes

```
import tensorflow as tf

model = tf.keras.Sequential()
model.add(tf.keras.layers.Input(shape=data.shape))
model.add(tf.keras.layers.Dense(512, activation='relu'))
model.add(tf.keras.layers.Dense(10, activation='softmax'))
optimizer = tf.keras.optimizers.SGD(lr=0.01)
loss = tf.keras.losses.categorical_crossentropy
model.compile(optimizer=optimizer, loss=loss)
```



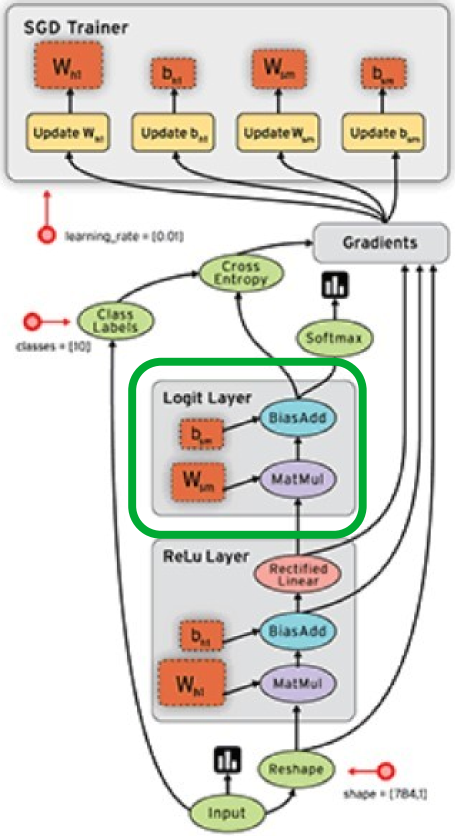


Multilayer perceptron in Keras

- Single hidden layer with 512 nodes

```
import tensorflow as tf

model = tf.keras.Sequential()
model.add(tf.keras.layers.Input(shape=data.shape))
model.add(tf.keras.layers.Dense(512, activation='relu'))
model.add(tf.keras.layers.Dense(10, activation='softmax'))
optimizer = tf.keras.optimizers.SGD(lr=0.01)
loss = tf.keras.losses.categorical_crossentropy
model.compile(optimizer=optimizer, loss=loss)
```

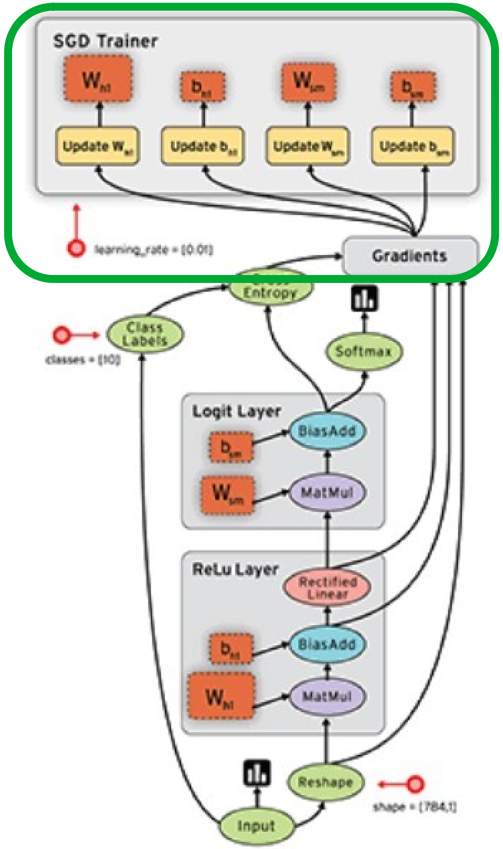


Multilayer perceptron in Keras

- Single hidden layer with 512 nodes

```python
import tensorflow as tf

model = tf.keras.Sequential()
model.add(tf.keras.layers.Input(shape=data.shape))
model.add(tf.keras.layers.Dense(512, activation='relu'))
model.add(tf.keras.layers.Dense(10, activation='softmax'))
optimizer = tf.keras.optimizers.SGD(lr=0.01)
loss = tf.keras.losses.categorical_crossentropy
model.compile(optimizer=optimizer, loss=loss)
```

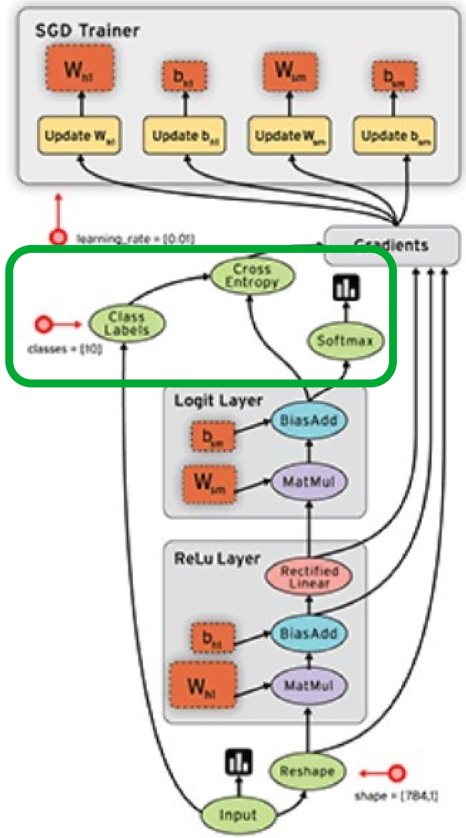


Multilayer perceptron in Keras

- Single hidden layer with 512 nodes

```
import tensorflow as tf

model = tf.keras.Sequential()
model.add(tf.keras.layers.Input(shape=data.shape))
model.add(tf.keras.layers.Dense(512, activation='relu'))
model.add(tf.keras.layers.Dense(10, activation='softmax'))
optimizer = tf.keras.optimizers.SGD(lr=0.01)
loss = tf.keras.losses.categorical_crossentropy
model.compile(optimizer=optimizer, loss=loss)
```

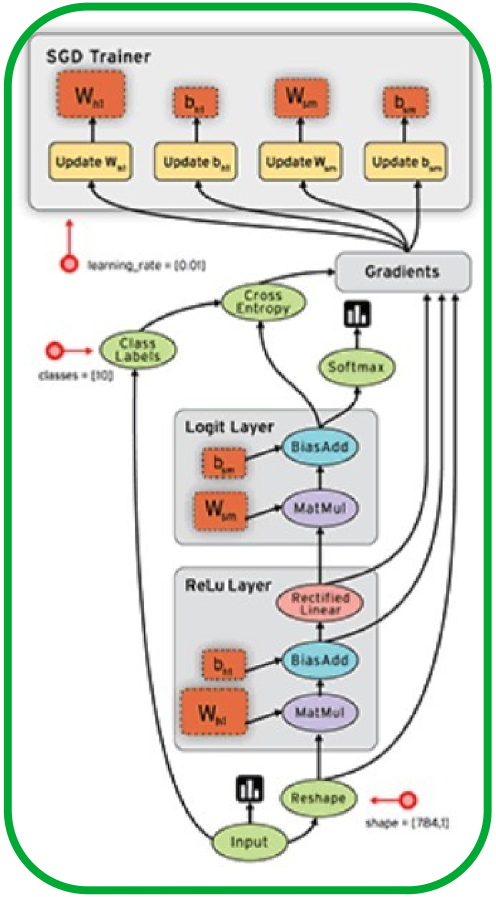


Multilayer perceptron in Keras

- Single hidden layer with 512 nodes

```
import tensorflow as tf

model = tf.keras.Sequential()
model.add(tf.keras.layers.Input(shape=data.shape))
model.add(tf.keras.layers.Dense(512, activation='relu'))
model.add(tf.keras.layers.Dense(10, activation='softmax'))
optimizer = tf.keras.optimizers.SGD(lr=0.01)
loss = tf.keras.losses.categorical_crossentropy
model.compile(optimizer=optimizer, loss=loss)
```

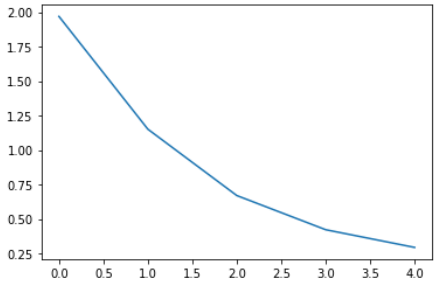


Training and evaluation

- Training on a single sample (to match the previous computational graph)

```
history = model.fit(data, labels, epochs=5)
plt.plot(history.history['loss'])
plt.show()
```

```
Train on 1 samples
Epoch 1/5
1/1 [==============================] - 0s 103ms/sample - los
s: 1.9689
Epoch 2/5
1/1 [==============================] - 0s 1ms/sample - loss:
1.1508
Epoch 3/5
1/1 [==============================] - 0s 1ms/sample - loss:
0.6696
Epoch 4/5
1/1 [==============================] - 0s 1ms/sample - loss:
0.4217
Epoch 5/5
1/1 [==============================] - 0s 1ms/sample - loss:
0.2933
```



- A lot of details taken care of by default:
  - Weight initialization
  - Label conversion
  - Loss functions
- Various details can be customized when defining models

Back to TensorFlow

- Keras is a part of Tensorflow 2.0 – an alternative and quick way of building traditional neural networks.
- When use Tensorflow instead of Keras (defining custom functions/objects yourself?)
  - Custom neural network models, layers, operations
  - Control over training/evaluation (instead of large default methods like *model.fit(), model.evaluate()*)
  - Research – developing novel architectures

TensorFlow

Numpy and TensorFlow analogies

- Basic API similarities:

NumPy

```
a = np.zeros([2, 3])
```

```
array([[0., 0., 0., 0., 0., 0.]])
```

```
a.reshape([1, 6])
```

```
array([[0., 0., 0., 0., 0., 0.]])
```

```
array([[0., 0., 0., 0., 0., 0.]])
```

```
np.arange(5)
```

```
array([0, 1, 2, 3, 4])
```

TensorFlow

```
b = tf.zeros([2, 3])
```

```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[0., 0., 0.],
       [0., 0., 0.]], dtype=float32)>
```

```
tf.reshape(b, [1, 6])
```

```
<tf.Tensor: shape=(1, 6), dtype=float32, numpy=arra
y([[0., 0., 0., 0., 0., 0.]], dtype=float32)>
```

```
tf.range(5)
```

```
<tf.Tensor: shape=(5,), dtype=int32, numpy=array([0, 1, 2, 3, 4], dty
pe=int32)>
```

- **PyTorch** – very similar API

Numpy and TensorFlow analogies

- Operations

NumPy

TensorFlow

```
np.dot(a, a.T)
np.mean(a, axis=0)
np.sum(a, axis=1)
np.argmax(a)
```

```
tf.matmul(b, tf.transpose(b))
tf.math.reduce_mean(b, axis=0)
tf.math.reduce_sum(b, axis=1)
tf.math.argmax(a)
```

- Tensorflow → Numpy object conversion

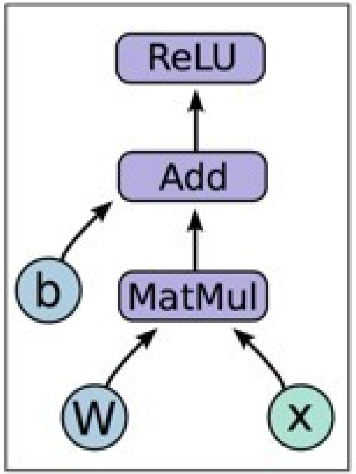
```
tf.zeros([2, 3]).numpy()
```

```
array([[0., 0., 0.],
       [0., 0., 0.]], dtype=float32)
```

Perceptron layer example

- Weights of a fully connected layer - 2nd rank Tensor

$$h = ReLU(Wx + b)$$



- **x** = [$N_{samples}$ * $F_{features}$] - *batch of input data*
- **W** = [$F_{features}$ * M_{nodes}] - *layer weights*
- **B** = [1 * M_{nodes}] - *layer biases*
- **Out** = [$N_{samples}$ * M_{nodes}]
- *Example*: 100 samples of MNIST, 512 node perceptron layer
- [100 x 784] x [784 x 512] → Output = [100 x 512]
  Activations of each node for each data sample

Perceptron layer in TensorFlow

- Layer with 512 nodes/units

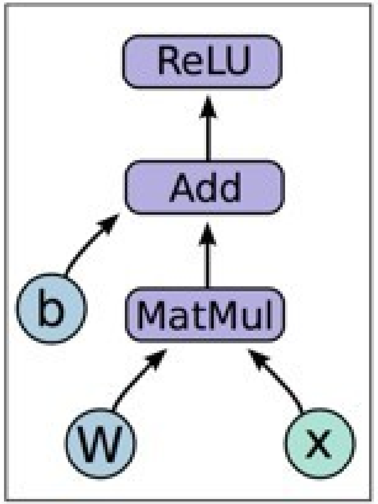
$$h = ReLU(Wx + b)$$

```
x.shape
```

```
(1, 784)
```

```
W = tf.random.normal([784, 512])
b = tf.ones([512])
product = tf.matmul(x, w) + b    # MatMul and Add operations
ReLU = tf.math.maximum(product, 0,)
ReLU.shape
```

```
TensorShape([1, 512])
```

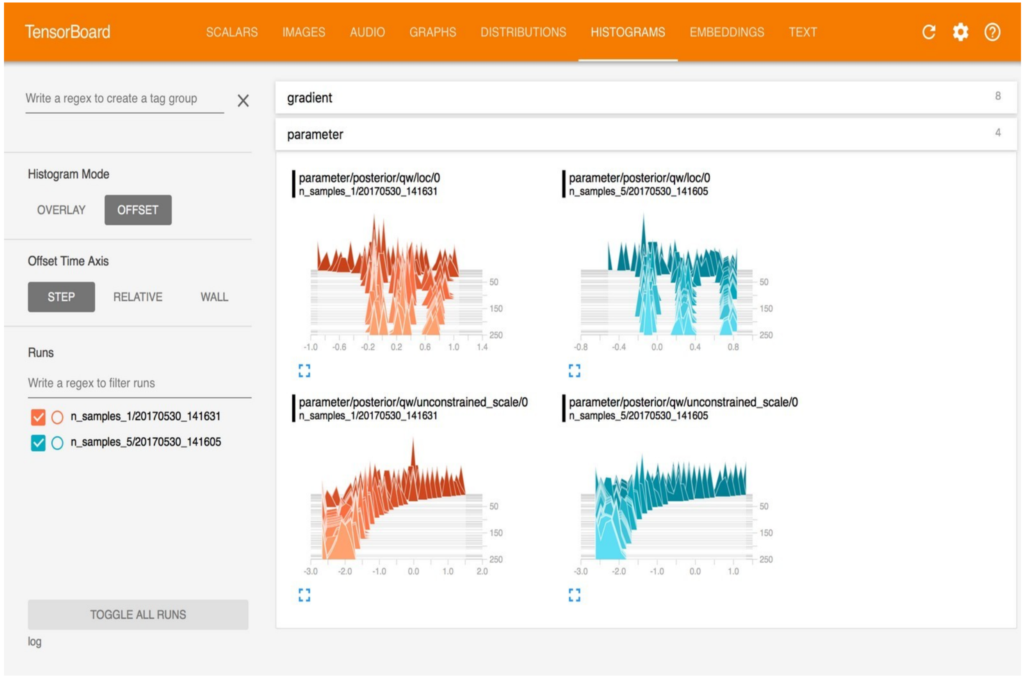


TensorBoard

- Track training metrics while training
- Visualize the computational graph

- Compare model iterations
- Track variables (weights, gradients, activation maps of convolutional layers)

- Takes care of most aspects of model monitoring (most of the time there is no need to make custom functions), well optimized



TensorBoard in Jupyter notebooks

- Load the module

```
%load_ext tensorboard
```

- Initialize the TensorBoard callback object

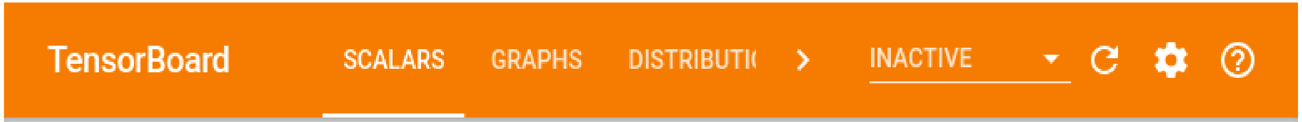
```
tensorboard_callback = tf.keras.callbacks.TensorBoard('logs', histogram_freq=1)
```

- Pass it as *callbacks* keyword argument for the .fit() method.

```
history = model.fit(x_train, y_train,
            epochs=10,
            callbacks=[tensorboard_callback])
```

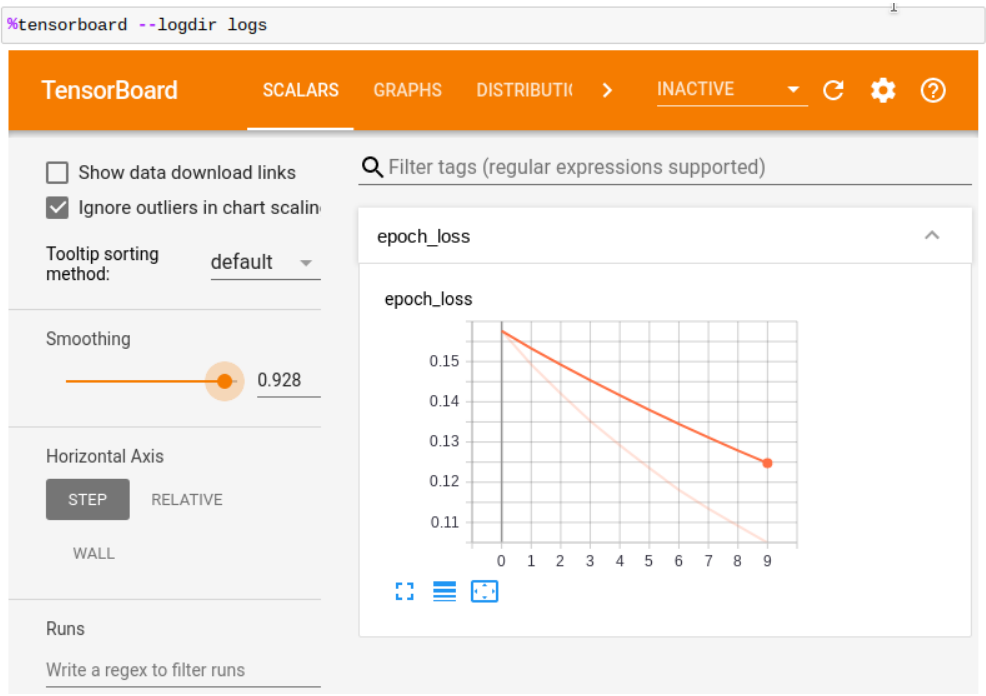
- Launch tensorboard within the notebook

```
%tensorboard --logdir logs
```

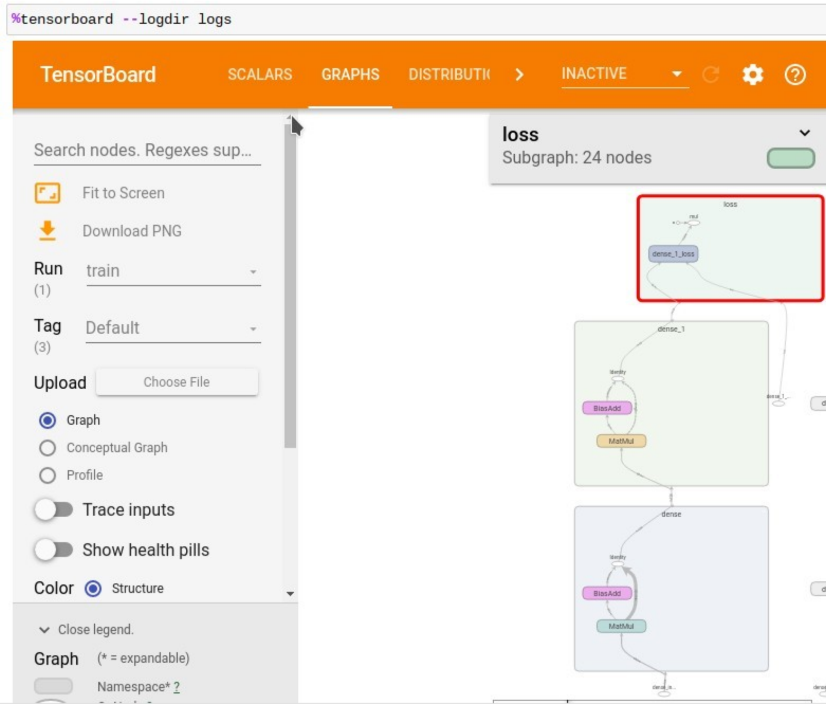


TensorBoard (2)

- Track metrics



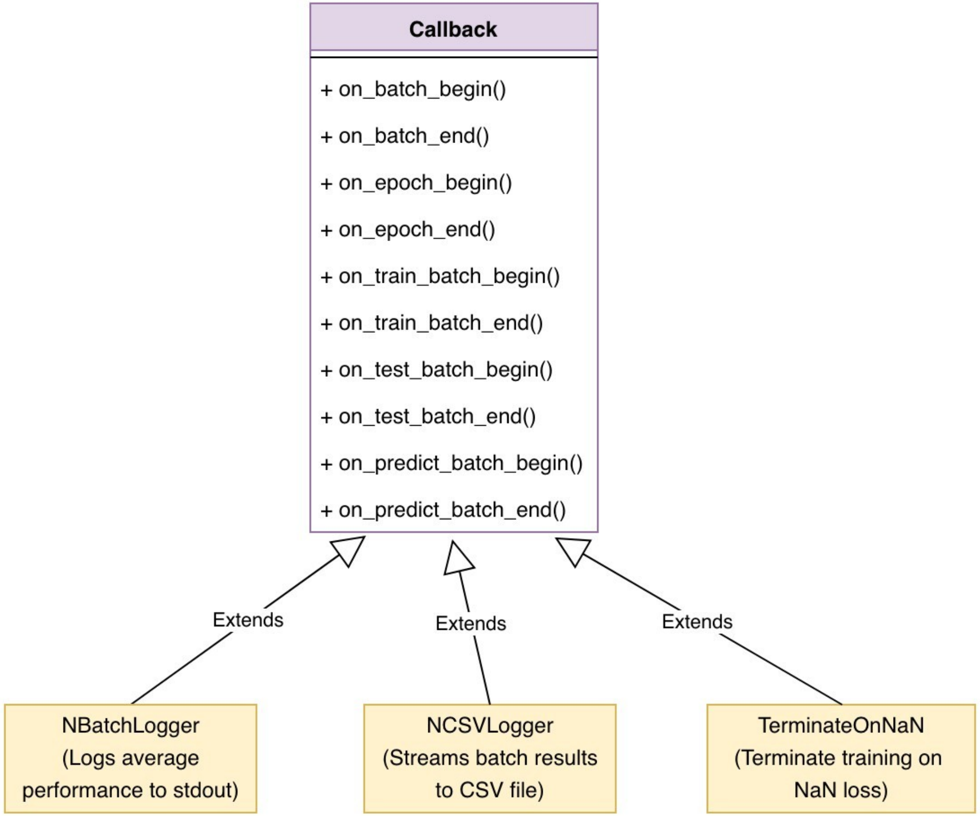
- Visualize computational graph



- Very easy to extend through other callbacks – track multiple metrics

Callbacks

- Modify the behavior of *model.fit(), model.evaluate() methods*
- Pass a custom callback object/function that gets called during each batch/epoch
- Inspect / retrieve the model variables while training
- Logs, checkpoints, saving
- Change hyperparameters during training (e.g. learning rate) based on current metrics



Custom callbacks

- Custom callbacks can be created by inheriting from the tf.keras.callbacks.Callback superclass.

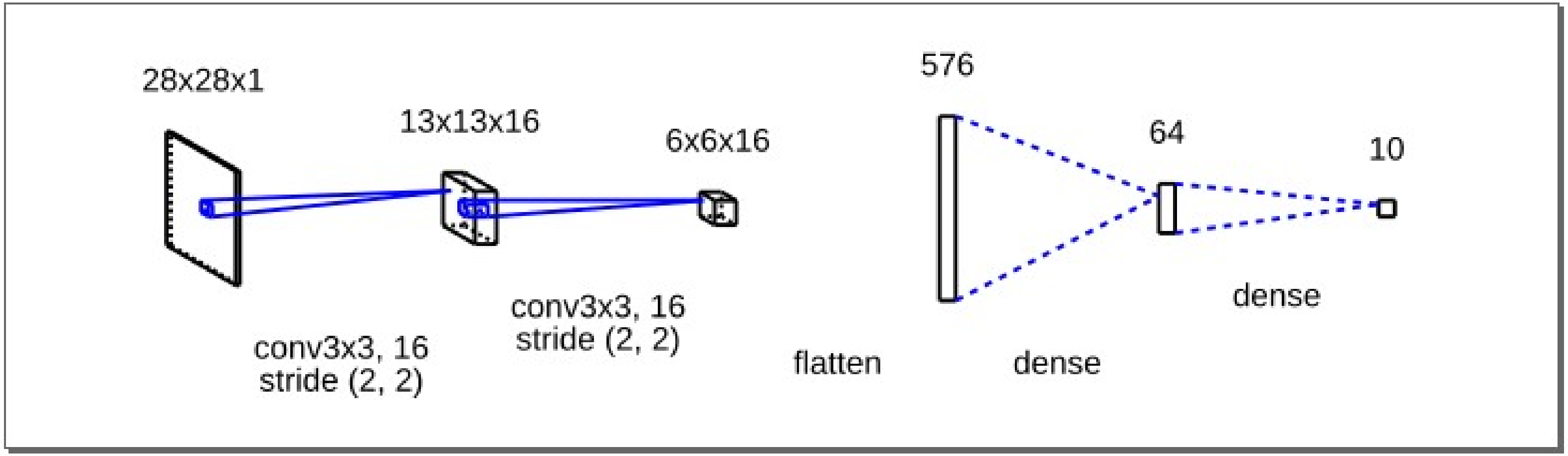
In [34]:
```
# A simple callback for printing weight values
class PrintCallback(tf.keras.callbacks.Callback):
    def on_epoch_begin(self, epoch, logs=None):
        print('Epoch: ', epoch)

    # Print the means of weights of the output layer
    def on_train_batch_begin(self, batch, logs=None):
        weights = model.layers[-1].get_weights()[0]
        weights = np.mean(weights, axis=0)
        print('  Batch: {}, mean weights: {}'.format(batch, np.round(weights, 3)))
```

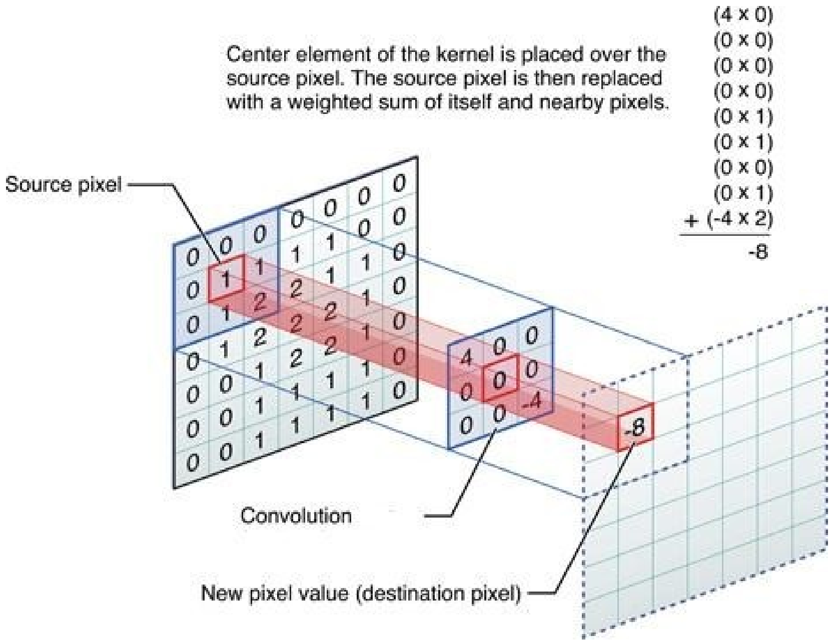
- If you override one of the following methods, your custom code will be executed during runtime.

Building Convolutional networks in Keras

```
conv_net = tf.keras.Sequential()
conv_net.add(tf.keras.layers.Input(shape=[28, 28, 1]))
conv_net.add(tf.keras.layers.Conv2D(filters=16, kernel_size=(3, 3), strides=(2, 2), activation='relu'))
conv_net.add(tf.keras.layers.Conv2D(filters=16, kernel_size=(3, 3), strides=(2, 2), activation='relu'))
conv_net.add(tf.keras.layers.Flatten())
conv_net.add(tf.keras.layers.Dense(64, activation='relu'))
conv_net.add(tf.keras.layers.Dense(10, activation='softmax'))
optimizer = tf.keras.optimizers.SGD(learning_rate=0.001)
loss = tf.keras.losses.categorical_crossentropy
conv_net.compile(optimizer=optimizer, loss=loss)
```



Tensors in a convolutional layer

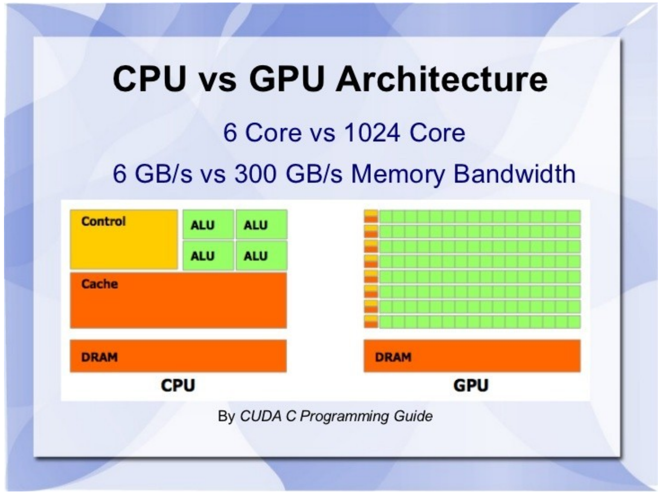


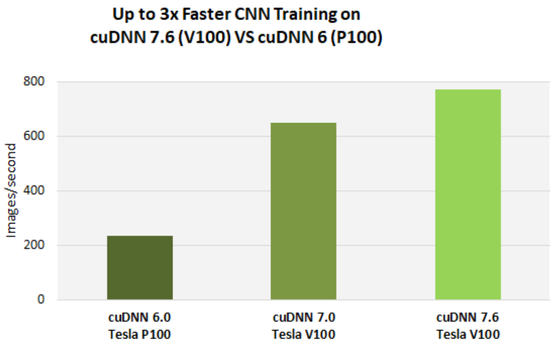
- Convolutional layers in Tensorflow involve 4D tensors:
  - [**N**samples * **X**in * **Y**in * **C**hannels] - Batch of images
  - **[C**hannels * **X**k * Yk * **F**ilters] - Filters / kernels (learnable weights)
  - Example: input consisting of 100 grayscale images of size 7x7, convolutional layer with 16 kernels of size 3x3, no padding
  - [100 x 7 x 7 x 1] @ [1 x 3 x 3 x 16]
    How to get the dot product?
    - Take a slice of the input tensor that has equal size in (2^{nd} and 3^{rd} axes) to the kernel
    - Output[x, y] = [**100** x 3 x 3 x 1] @ [1 x 3 x 3 x **16**] = [**100** x **16**]
    - Repeat for each patch of the image → [100 x 5 x 5 x 16]

- Expensive computationally, but has less trainable parameters

CPUs, GPUs, TPUs

- GPU faster memory interfaces, higher bandwidth
- Serial vs parallel execution
- CuDNN, neural networks:
  - Mostly of tensor operations - parallelism
  - Require high number of FLOPS (expensive operations, e.g. convolutions)
  - Using GPU Faster, lower power consumption
- Tensorflow - not only for neural networks – good for parallelizing various algorithms
- GPUs – Nvidia vs AMD. Nvidia leads both in hardware and software (for deep learning).

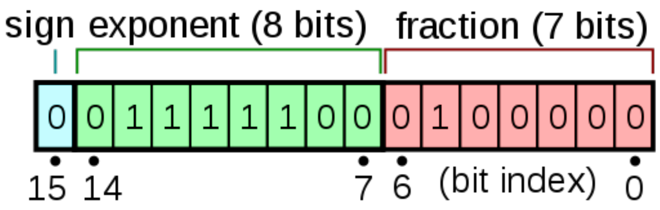


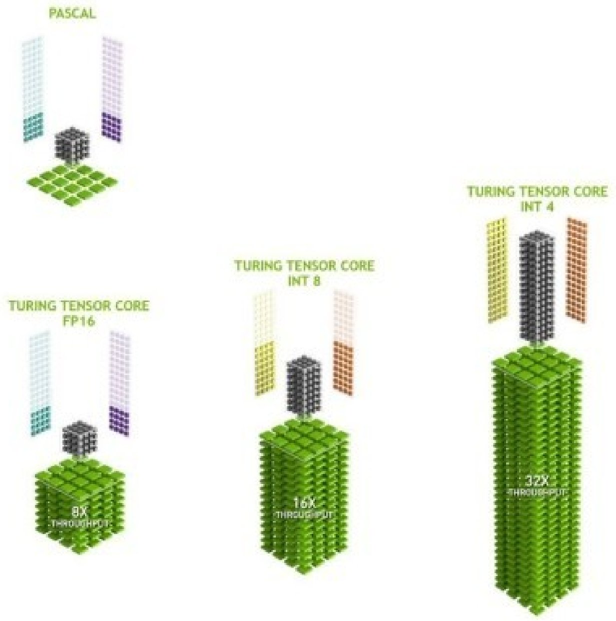


TensorFlow performance (images/sec), Tesla P100 + cuDNN 6 (FP32) on 17.12 NGC container, Tesla V100 + cuDNN 7.0 (Mixed) on 18.02 NGC container, Telsa V100 + cuDNN 7.6 (Mixed) on 19.05 NGC container, ResNet-50, Batch Size: 128

Tensor data types

- Various ways to represent data:
  - FP64, FP32, FP16 (number of bits per floating point number) – speed/performance trade-off:
    - FP32 (float32) – standard ($1e^{-38}$ – $3e^{38}$ range)
    - FP64 – high precision (scientific applications, only high-end GPUs support this format)
    - FP16 – fast training (good for initial development stages). Nvidia RTX GPUs have FP16 support.
  - INT8, INT4 (8 or 4 bit integers) – inference speedup (after training)
    - Often neural network input data is inherently noisy
    - Therefore a lot of research into less precise data representations without compromising on performance (energy savings, performance)
    - Great for real-time applications (e.g. video classification)





Questions?

- Assignments:
  - A1 – last lab session today
  - A2 – to be published next week