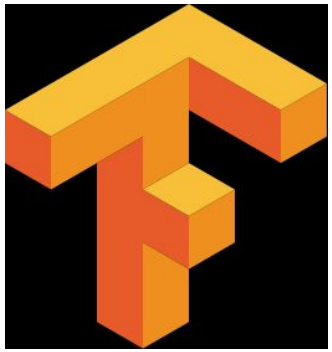
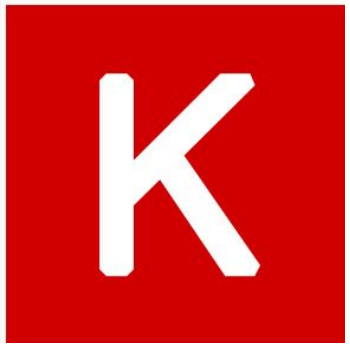


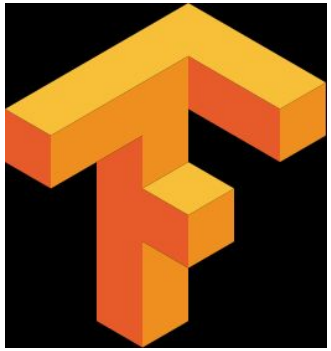


An introduction to Tensorflow and Keras

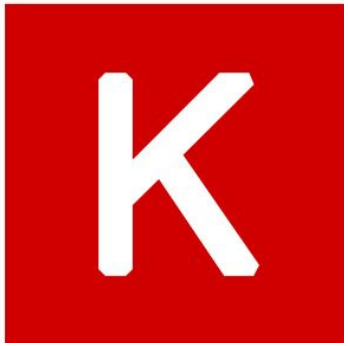
Intelligent Systems for Pattern Recognition (ISPR)

Luca Pedrelli
Postdoctoral Researcher at
Department of Computer Science, Università di Pisa
luca.pedrelli@di.unipi.it

Deep Learning Frameworks



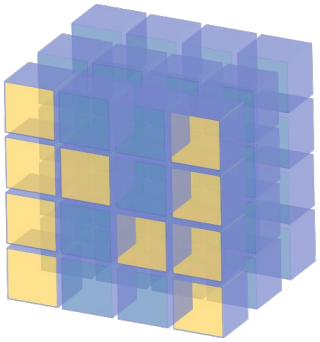
- **Data Flow Graph**
  of tensor operations
  - ➢ **Automatic differentiation**
  - ➢ High degree of **parallelization**



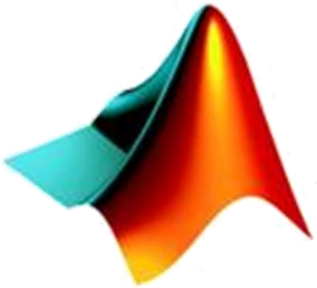
- High-level **Object-oriented** interface
  for other backand libraries:
  - ➢ **Tensorflow**, Theano, CNTK.

Why we should use Tensorflow and Keras?

Before Deep Learning Frameworks
(Linear Algebra and Numerical Analysis)



NumPy

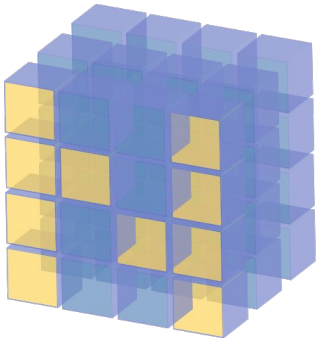


MATLAB

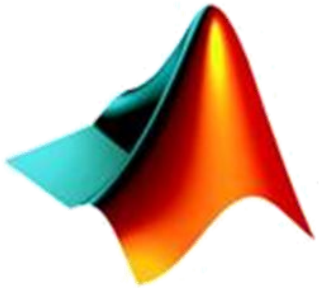
Powerful libraries for linear algebra, numerical analysis and statistics:

- Very efficient implementation for matrix and tensor operations:
  - **dot product, tensordot, sum, multiplication, ...**
- Very efficient implementation for linear algebra operations:
  - **matrix factorizations, eigenvalue decomposition, SVD, cholesky, QR, …**
- Functional programming:
  - easy data aggregation with **lambda expressions, map, reduce, filter, …**
  - easy funcion definition and **composition**

Understanding Deep Learning Frameworks stemming from NumPy and Matlab



NumPy



MATLAB

Drawbacks for Deep Learning design:

- No general **design tools** for DL
- No **automatic differentiation**
- **GPU parallelization** by hand
- No **general parallelization** depending on function structure !

However:

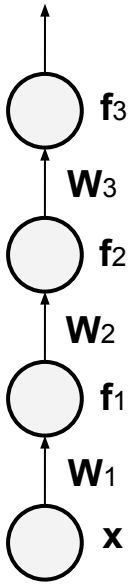
- **NumPy functionalities** (**numpy.arrays**) are widely used in most python implementations for machine learning approaches (also in **Tensorflow** and **Keras**)
- Numpy and Matlab are very useful to understand the design of Neural Network models. (very intuitive syntax to implement matrix operations and functions definition)
- Tensorflow operations are very inspired by NumPy operations

Design of Deep Neural Networks without DL frameworks

- Define the **model function (i.e., the NN architecture)**:
  - composition of differentiable operations
- Define the **loss function (to minimize or maximize)**:
  - differentiable function defined on the basis of model functon and training data
- Compute **analytically** the **derivative** of the loss
- Implement the **forward pass** basing on the **model function**
- Implement the **backward pass** basing on the **derivative** of the loss (i.e., back-propagation)
- Implement the **gradient descent** approach

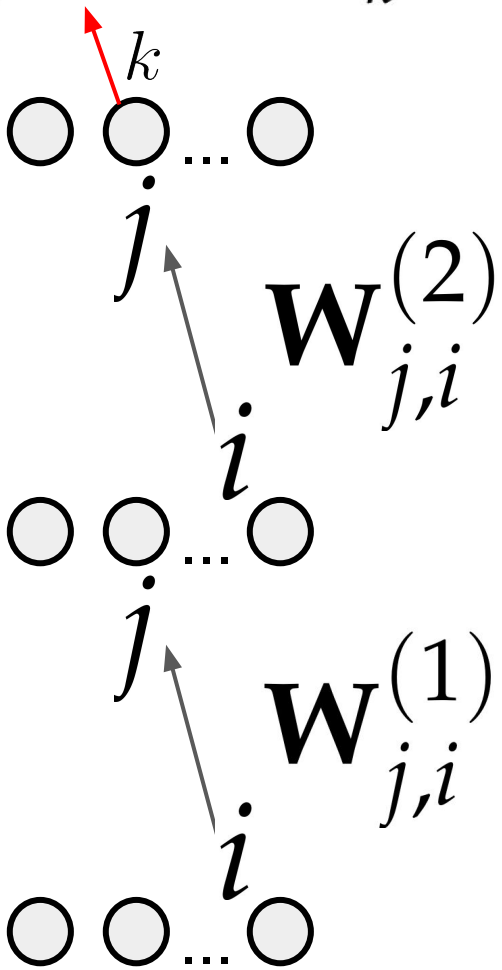
MLP architecture example

$$y = f_3(W_3\, f_2(W_2\, f_1(W_1 x)))$$



Design of NN Architecture

$$O_k^{(2)} = f(net_k^{(2)}) = f(\sum_z \mathbf{W}_{k,z}^{(2)} O_z^{(1)})$$



$\mathbf{W}_{j,i}^{(2)}$

$\mathbf{W}_{j,i}^{(1)}$

Training Set for a Supervised Learning Task

$$\{(x^1, d^1), (x^2, d^2), ..., (x^T, d^T)\}$$

$$x^p \in \mathbb{R}^{N_{inputs}}$$

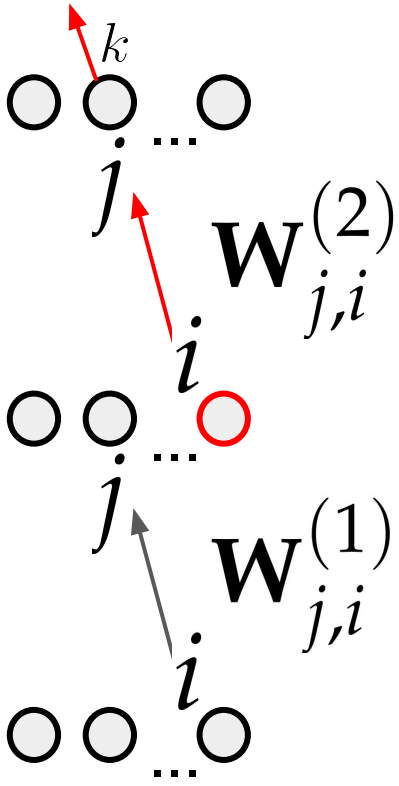
$$d^p \in \mathbb{R}^{N_{targets}}$$

$$E_{tot} = \sum_p E_p$$

$$E_p = \sum_z \frac{1}{2}(d_z - O_z^{(2)})^2$$

Delta Rule for Top Layer

$$O_k^{(2)} = f(net_k^{(2)}) = f(\sum_z \mathbf{W}_{k,z}^{(2)} O_z^{(1)}) \qquad E_{tot} = \sum_p E_p$$



$$E_p = \sum_z \frac{1}{2}(d_z - O_z^{(2)})^2$$

$\mathbf{W}_{j,i}^{(2)}$

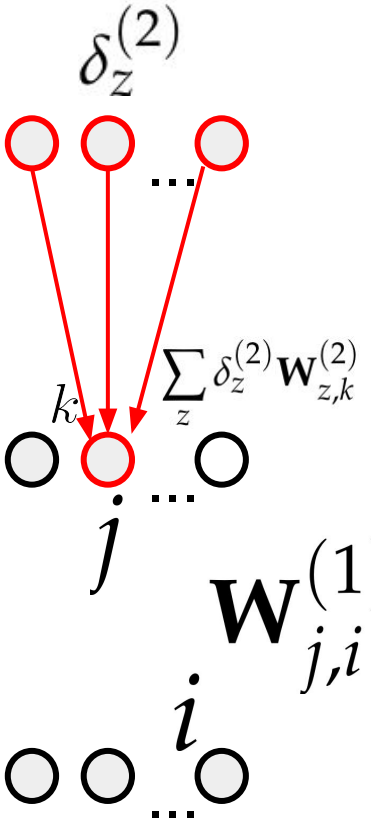
$\mathbf{W}_{j,i}^{(1)}$

$$-\frac{\partial E_p}{\partial \mathbf{W}_{\mathbf{k,i}}^{(2)}} = -\frac{\partial E_p}{\partial net_k^{(2)}} \frac{\partial net_k^{(2)}}{\partial \mathbf{W}_{\mathbf{k,i}}^{(2)}}$$

$$= (d_k - O_k^{(2)}) f'(net_k^{(2)}) \, O_i^{(1)}$$

$\delta_k^{(2)}$

Delta Rule for a generic Hidden Layer



$$E_p = \frac{1}{2}\sum_z (d_k - O_k^{(2)})^2 \quad -\frac{\partial E_p}{\partial \mathbf{W}_{\mathbf{k,i}}^{(\mathbf{1})}} =$$

$$\mathbf{W}_{j,i}^{(1)} \quad = -\frac{\partial \sum_z \frac{1}{2}(d_z - O_z^{(2)})^2}{\partial net_z^{(2)}} \frac{\partial net_z^{(2)}}{\partial \mathbf{W}_{\mathbf{k,i}}^{(\mathbf{1})}}$$

$$\sum_z \delta_z^{(2)} \qquad \delta_k^{(1)}$$

$$= \sum_z \delta_z^{(2)} \frac{\partial net_z^{(2)}}{\partial O_k^{(1)}} \frac{\partial O_k^{(1)}}{\partial net_k^{(1)}} \frac{\partial net_k^{(1)}}{\partial \mathbf{W}_{k,i}^{(1)}} = \sum_z \delta_z^{(2)} \mathbf{W}_{z,k}^{(2)} f'(net_k^{(1)}) O_i^{(0)}$$

MLP: training algorithm

for epoch in epochs:
    for p in training_set:
        for l in L-1...0:
            for k ...:

$$f = \tanh(x) \quad f' = 1 - \tanh(x)^2$$

If Output Layer

$$\delta_k^{(l)} = (d_k - O_k^{(l)}) f'(net_k^{(l)})$$

Otherwise

$$\delta_k^{(l)} = \sum_z \delta_z^{(l+1)} \mathbf{W}_{z,k}^{(l+1)} f'(net_k^{(l)})$$

for i ...:

$$-\frac{\partial E_p}{\mathbf{W}_{k,i}^{(l)}} = \delta_k^{(l)} O_i^{(l-1)}$$

        for l in 0...L-1:

$$\mathbf{W}^{(l)} = \mathbf{W}^{(l)} + lr\Delta\mathbf{W}^{(l)}$$

MLP: NumPy implementation (An example) network initialization

```
# Create network: initialize weights from [-1,1]
Nlayers = 2
Neurons = 100

W = []
layers = range(Nlayers)
for l in layers:
      if len(layers) == 1:
            W.append(np.random.uniform(-1,1, (Noutputs, Ninputs+1)))
      elif l == 0:
            W.append(np.random.uniform(-1,1, (Neurons, Ninputs+1)))
      elif l < Nlayers-1:
            W.append(np.random.uniform(-1,1, (Neurons, Neurons+1)))
      else:
            W.append(np.random.uniform(-1,1, (Noutputs, Neurons+1)))
```

MLP: NumPy implementation (An example) forward pass

```
# Compute layers outputs
def forward_pass(W, inputs):
      otuputs = []
      for l in range(len(W)):
            if l==len(W)-1:
                  otuputs.append(W[l][:,:-1].dot(inputs) + np.expand_dims(W[l][:,-1], axis=1))
            else:
                  otuputs.append(np.tanh(W[l][:,:-1].dot(inputs) + np.expand_dims(W[l][:,-1], axis=1)))
            inputs = otuputs[-1]

      return otuputs
```

MLP: NumPy implementation (An example) backward pass

```
# Compute a gradient descent pass on a sample input
def backward_pass(W, sample_input, single_target):

	outputs = forward_pass(W, sample_input)

	layers_gradients = []
	deltas = []
	dWs = []

	for l in reversed(range(len(W))):
		delta = np.zeros(W[l].shape[0])
		dW = np.zeros(W[l].shape)

		for k in range(W[l].shape[0]):
			if l == Nlayers-1:
				delta[k] = (single_target[k,0] - outputs[l][k,0])

			else:
				delta_kk = 0.0
				for z in range(deltas[-1].shape[0]):
					delta_kk = delta_kk + deltas[-1][z] * W[l+1][z,k]

				delta[k] = delta_kk * (1 - outputs[l][k,0]**2)

			for i in range(W[l].shape[1]-1): # without bias
				if l==0:
					dW[k,i] = delta[k] * sample_input[i,0]
				else:
					dW[k,i] = delta[k] * outputs[l-1][i,0]

			dW[k,-1] = dW[k,-1] + W[l][k,-1] # add bias

		deltas.append(delta)
		dWs.append(dW)

	return dWs
```

MLP: NumPy implementation (An example)
Online/Batch Gradient Descent

```
# leraning rate
lr = 0.0001
epochs = 20
errors = []

for epoch in range(epochs):
      print(epoch)
      for p in Full_TR_indexes:
            dWs = backward_pass(W, inputs[:,p:p+1], targets[:,p:p+1])

            # online GD
            for l in reversed(range(len(W))):
                  W[Nlayers-1-l] = W[Nlayers-1-l] + lr*dWs[l]

      outputs = forward_pass(W, inputs)[-1]
      errors.append(MSE(outputs[:, Full_TR_indexes], targets[:, Full_TR_indexes]))
```

- Stochastic gradient descent with mini-batch approach?
- Parallel computing?
- Other gradient descent approaches? (weight decay, momentum…)
- …

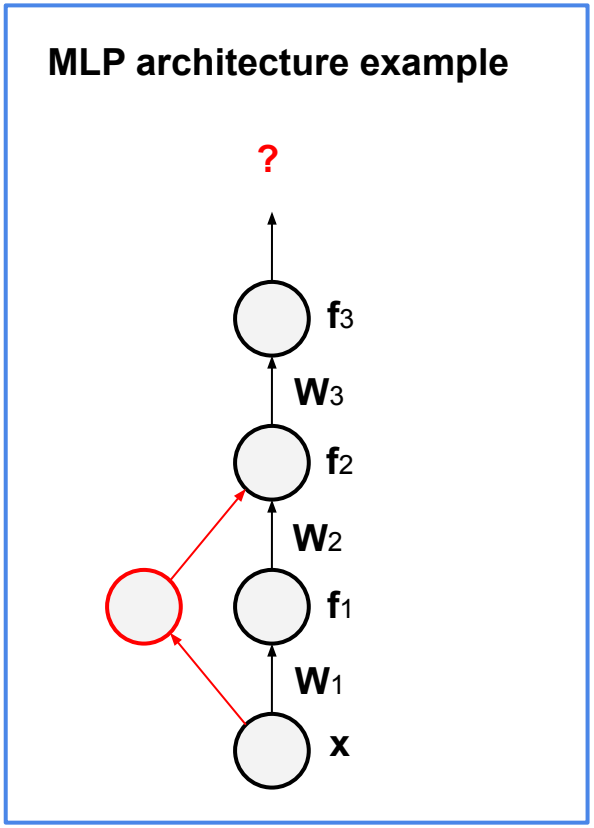
Design of Deep Neural Networks without DL frameworks (Conclusions)

Overall, if we change the topology of the network we need to recompute the **derivative** and to recode the **backward function**!

- Compute **analytically** the **derivative** of the loss ?
- Implement the **forward pass** basing on the **model function** ?
- Implement the **backward pass** basing on the **derivative** of the loss (i.e., back-propagation) ?

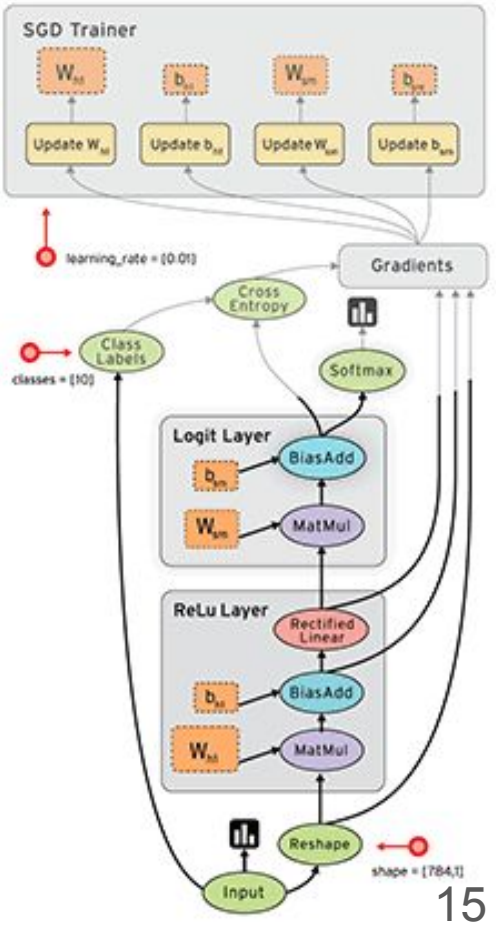
Two Solutions:

Tensorflow and Keras allow us to automatically compute the **derivative** and the **backward pass** just defining the **forward pass ! (i.e., the network architecture)**

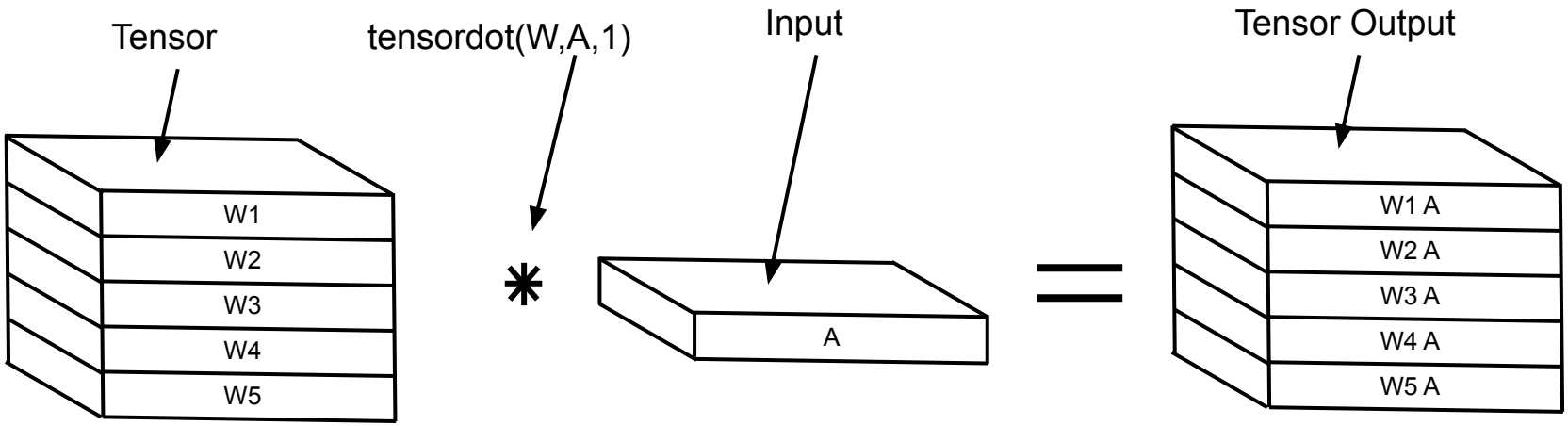


TensorFlow

- Define NN models by means of **tensor** operations (dot product, tensor contraction, sum, multiplication, ...)
- Models represented as a static **Data Flow Graph**
- **Automatic differentiation** based on the Graph! We avoid to implement the backward pass!
- **Automatic parallelization** based on the Graph! Both for CPU and GPU!



Tensor

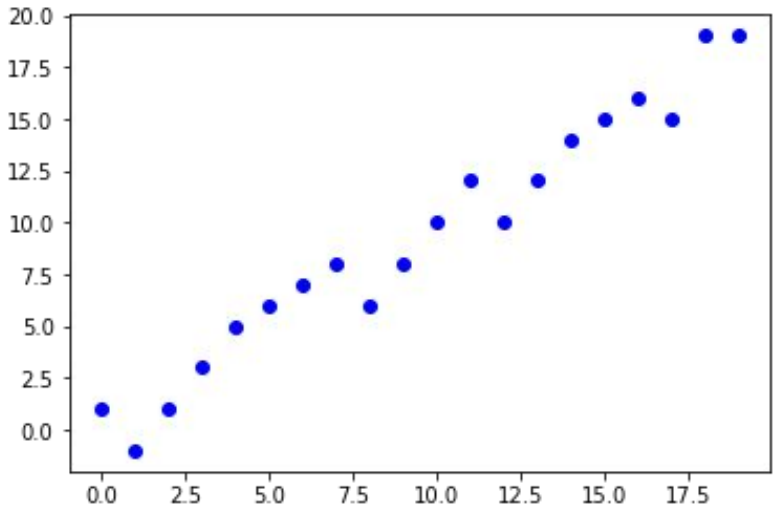


- A Tensor is a geometric object that can be defined in several ways depending on the level of abstraction (Algebra, Geometry, Physics, ...)

- From a Computer Science point of view we can see a Tensor as:
  - A multi-dimensional array:
    - Useful to represent data
  - A multi-linear map (i.e., tensordot, tensor contraction):
    - Useful to parallelize computation on **GPU** and **CPU**

Linear Regression (A simple example)

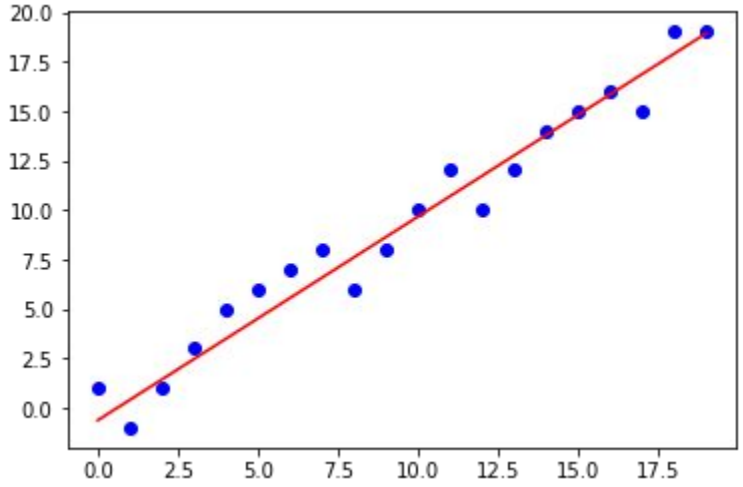
```
import numpy as np
import matplotlib.pyplot as plt

# Generating data
X = np.arange(20)
Y = X + np.random.randint(-2, 2, X.shape[0])

# Visualizing
plt.figure()
plt.scatter(X, Y, c='b')
plt.show()
```




```
import tensorflow as tf

# Defining computation graph
W = tf.Variable(tf.random_normal([1]), name='weight')
b = tf.Variable(tf.random_normal([1]), name='bias')
x = tf.placeholder(tf.float32, shape=[None])
y = tf.placeholder(tf.float32, shape=[None])

# Our hypothesis x*W+b
hypothesis = x * W + b

# Cost/loss function (mean squared error)
cost = tf.reduce_mean(tf.square(hypothesis - y))

# Minimize
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
minimize = optimizer.minimize(cost)
```

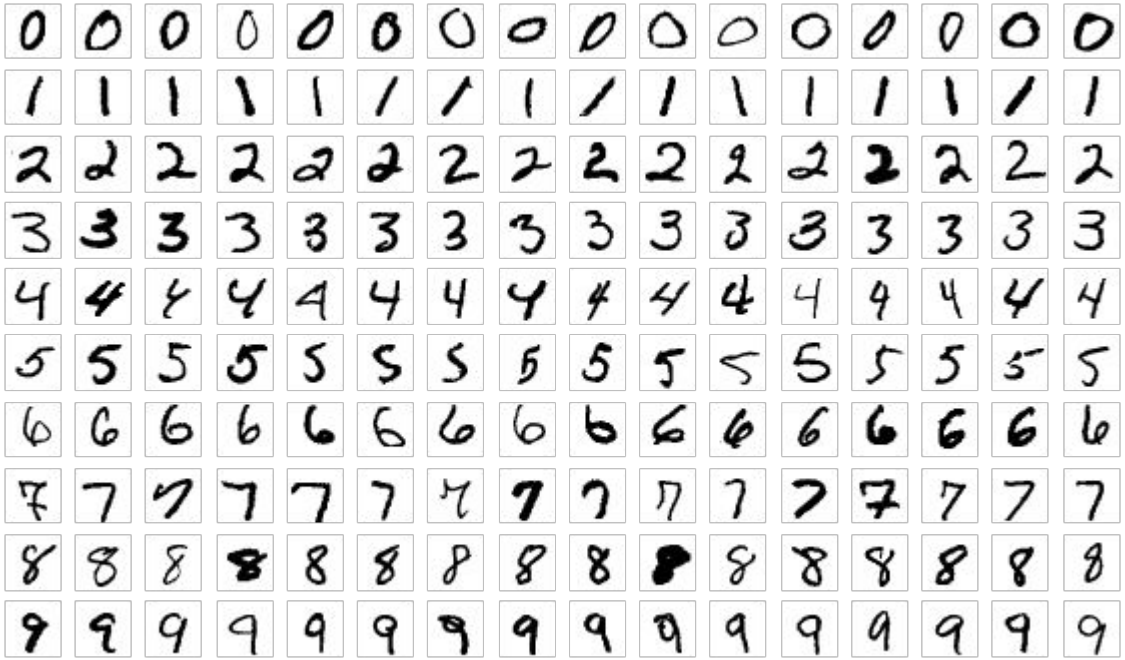
Linear Regression 2 (A simple example)

```
# Create a Session and fit the model to data
sess = tf.Session()

# Initializes global variables in the graph.
sess.run(tf.global_variables_initializer())

# Fit the line with new training data
for epochs in range(300):
      cost_val, W_val, b_val, _ = sess.run([cost, W, b, minimize], feed_dict={x: X, y: Y})

plt.figure()
plt.scatter(X, Y, c='b')
plt.plot(W_val*range(X.shape[0]) + b_val, c='r')
plt.show()
```



MINST (a dataset of handwritten digits)

- Image **classification**
- A baseline task for computer vision
- Very useful to evaluate and design basic DL models for computer vision



Multi Layer Perceptron (MLP) in Tensorflow

```
# Read data
mnist = mnist_data.read_data_sets("data", one_hot=True, reshape=False, validation_size=0)

# Defining the computation graph
# ================ MODEL ================
X = tf.placeholder(tf.float32, [None, 28, 28, 1])
XX = tf.reshape(X, [-1, 784])

# Input2Hidden layer
W1 = tf.Variable(tf.truncated_normal([784, 256]))
b1 = tf.Variable(tf.zeros([256]))
h1 = tf.nn.relu(tf.matmul(XX, W1) + b1)

# Hidden2Hidden layer
W2 = tf.Variable(tf.truncated_normal([256, 256]))
b2 = tf.Variable(tf.zeros([256]))
h2 = tf.nn.relu(tf.matmul(h1, W2) + b2)

# Hidden2Output layer
W3 = tf.Variable(tf.zeros([256, 10]))
b3 = tf.Variable(tf.zeros([10]))
Y = tf.matmul(h2, W3) + b3  # ← Y.shape() = (None, 10) (!)
# =======================================
```

Cross entropy loss (typically used for classification)

```
# Loss and optimizer
Y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=Y_, logits=Y))
train_step = tf.train.AdamOptimizer(0.001).minimize(cross_entropy)

# Evaluate model
correct_prediction = tf.equal(tf.argmax(Y, 1), tf.argmax(Y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# Train the model
N_EPOCHS = 10
BATCH_SIZE = 100
N_BATCHES = int(mnist.train.num_examples / BATCH_SIZE)

with tf.Session() as sess:
       sess.run(tf.global_variables_initializer())
       # Train
       for i in range(N_EPOCHS):
           for _ in range(N_BATCHES):
              batch_xs, batch_ys = mnist.train.next_batch(BATCH_SIZE)
              sess.run(train_step, feed_dict={X: batch_xs, Y_: batch_ys})

       # Evaluate accuracy on test set
       acc = sess.run(accuracy, feed_dict={
                     X: mnist.test.images,
                     Y_: mnist.test.labels
                     })

       print("Epoch {0}: Test accuracy --> {1:.2f}%".format(i, acc * 100))
```

...
Epoch 5: Test accuracy --> 95.40%
Epoch 6: Test accuracy --> 95.75%
Epoch 7: Test accuracy --> 95.77%
Epoch 8: Test accuracy --> 96.16%
Epoch 9: Test accuracy --> 95.83%

Convolutional Neural Networks (CNNs) in Tensorflow

```
# Helpers
def weight_variable(shape):
	initial = tf.truncated_normal(shape, stddev=0.1)
	return tf.Variable(initial)
def bias_variable(shape):
	initial = tf.constant(0.1, shape=shape)
	return tf.Variable(initial)
# Convolutions and poolings
def conv2d(x, W):
	return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')
def max_pool_2x2(x):
	return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],strides=[1, 2, 2, 1], padding='SAME')

# Defining the computation graph
# ================ MODEL ================
X = tf.placeholder(tf.float32, [None, 28, 28, 1])
# First convolutional layer
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
h_conv1 = tf.nn.relu(conv2d(X, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)
# Second convolutional layer
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)
# Fully-connected layer
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])
h_pool2_flat = tf.reshape(h_pool2, [-1, 7 * 7 * 64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
# Readout layer
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])
Y = tf.matmul(h_fc1, W_fc2) + b_fc2
# =======================================
```

TensorFlow

- Fast design of DL models! **MLP** 95.83% of accuracy -> **CNN** 98.78% of accuracy
- We only need to recode the model structure!
- Automatic parallelization! Very fast computation!

- Drawbacks:
  - Fast design only for simple models:
    Dropout? **BatchNormalization**? **Early stopping**? **RNNs**?
    ...
  - No support for object-oriented coding:
    The **model extension** is difficult

Keras

- High-level Object-oriented interface for other backand libraries:
  - **Tensorflow**, Theano, CNTK.

- Define NN models by means of **layer** operations (Dense, Conv1D, Conv2D, MaxPooling2D, Flatten, ...)

- Models represented as a stack of **layers**

Layer (type)	Output Shape
input_1 (InputLayer)	(None, 784)
dense_1 (Dense)	(None, 256)
dense_2 (Dense)	(None, 256)
dense_3 (Dense)	(None, 10)

- **Automatic differentiation** based on the Graph!
  We avoid to implement the backward pass!

- Great support for **Obect-oriented** coding**:**
  - You can extend functionalities extending classes:
    - Model class, Layer class, ...

Keras

Three ways to define a DL model

- Sequential model
  - stack of layers
  - easy to use
- Functional API
  - function of functions
  - general graph topologies
- Object-oriented
  - model subclassing
  - make the model extendible

MLP in Keras

Sequential model

```
# ================ MODEL ================
model = Sequential()
model.add(Dense(256, activation='relu', input_shape=(784,)))
model.add(Dense(256, activation='relu'))
model.add(Dense(10, activation='softmax'))
# =======================================
```

Functional API

```
# ================ MODEL ================
inputs = Input(shape=(784,))
hidden = Dense(256, activation='relu')(inputs)
hidden = Dense(256, activation='relu')(hidden)
outputs = Dense(10, activation='softmax')(hidden)
model = Model(inputs = inputs, outputs = outputs)
# =======================================
```

Object-oriented

```
# ================ MODEL ================
class MLP(Model):
       def __init__(self, num_classes=10):
          super(MLP, self).__init__(name='MLP')
          self.dense_1 = Dense(256, activation='relu')
          self.dense_2 = Dense(256, activation='relu')
          self.dense_3 = Dense(num_classes, activation='softmax')

       def call(self, inputs):
          x = self.dense_1(inputs)
          x = self.dense_2(x)
          return self.dense_3(x)

model = MLP(10)
# =======================================
```

CNN in Keras

```
# ================ MODEL ================
inputs = Input(shape=(28,28,1))
hidden = Conv2D(32, kernel_size=(5, 5), activation='relu')(inputs)
hidden = MaxPooling2D(pool_size=(2, 2))(hidden)

hidden = Conv2D(64, kernel_size=(5, 5), activation='relu')(hidden)
hidden = MaxPooling2D(pool_size=(2, 2))(hidden)

hidden = Flatten()(hidden)
hidden = Dense(1024, activation='relu')(hidden)
outputs = Dense(10, activation='softmax')(hidden)
model = Model(inputs = inputs, outputs = outputs)
# =======================================
# ================ TRAINING ================
model.compile(loss='categorical_crossentropy',
           optimizer=Adam(),
           metrics=['acc'])

history = model.fit(x_train, y_train, validation_data = (x_test, y_test),
           batch_size=100,
           epochs=3,
           verbose=1)
# =======================================
```

Very few code lines!
Test accuracy on MNIST -> more that 99% in 3 epochs of training

RNN in Keras (Many-to-one)
Classify sequences

```
# ================ MODEL ================
model = Sequential()
model.add(Embedding(max_features, 128))
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))
# =======================================
```

RNN in Keras (Many-to-many)
Classify each step

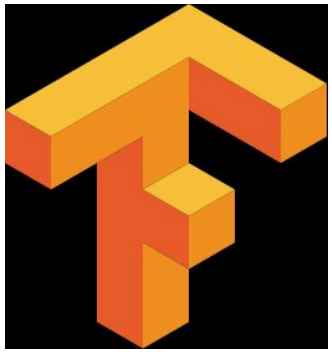
```
# ================ MODEL ================
model = Sequential()
model.add(Embedding(max_features, 128))
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2, return_sequences = True))
model.add(TimeDistributed(Dense(1, activation='sigmoid')))
# =======================================
```

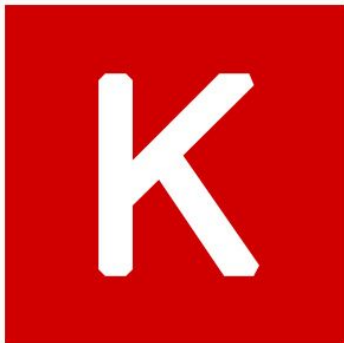
Deep RNNs

```
# ================ MODEL ================
model = Sequential()
model.add(Embedding(max_features, 128))
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2, return_sequences = True))
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))
# =======================================
```

Bidirectional RNNs

```
# ================ MODEL ================
model = Sequential()
model.add(Embedding(max_features, 128))
model.add(Bidirectional(LSTM(128, dropout=0.2, recurrent_dropout=0.2)))
model.add(Dense(1, activation='sigmoid'))
# =======================================
```





https://www.tensorflow.org/ https://keras.io/

See Documentations!