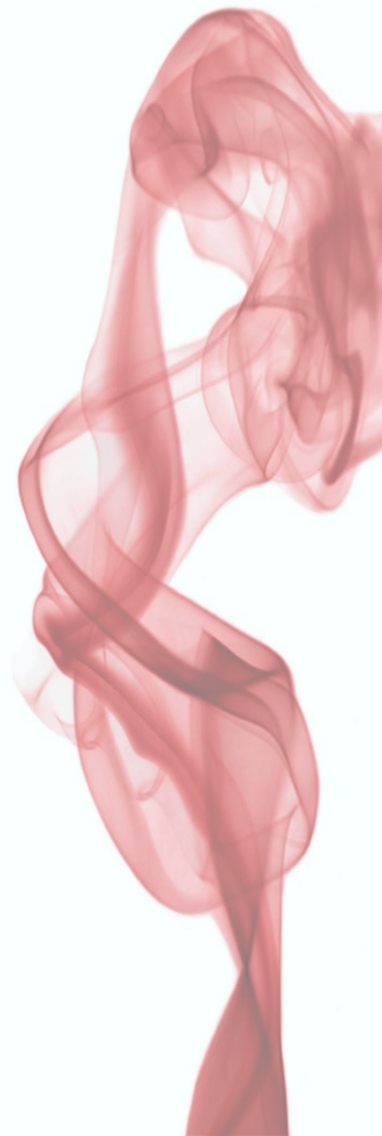


TMVA Implementation

Seungmok Lee

2020.02.05




Summary & Overview

- We decided to try TMVA in C and Python
- I implemented MNIST classifier in kDL/kPyKeras for exercise.
- Today, I'll explain how to implement DL with TMVA.
 - All codes are available on my github.
 - <https://github.com/physmlee/DLStudy>

List of Codes

- **MNISTtoROOT.py** ————— Data format transformer
- MNISTTMVA.py
- MNISTTMVA.C
- MNISTTMVAApplication.C
- MNISTTMVAApplication.py
- MNIST_PyMVA_CNN.py
- MNIST_TMVA_CNN.C

List of Codes

- MNISTtoROOT.py
 - MNISTTMVA.py
 - MNISTTMVA.C
 - MNISTTMVAApplication.C
 - MNISTTMVAApplication.py
 - MNIST_PyMVA_CNN.py
 - MNIST_TMVA_CNN.C
- 
- Train

List of Codes

- MNISTtoROOT.py

- MNISTTMVA.py

- MNISTTMVA.C

Application

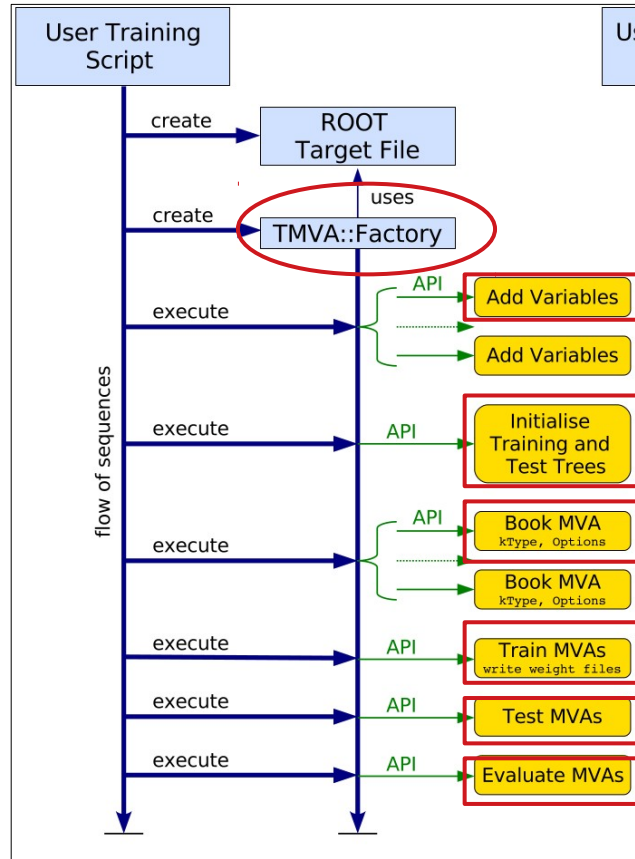
- MNISTTMVAApplication.C

- MNISTTMVAApplication.py

- MNIST_PyMVA_CNN.py

- MNIST_TMVA_CNN.C

TMVA Flow Chart



MNISTTMVA.py

```
# Model generating start
```

```
model = Sequential()
```

```
model.add(Dense(512, input_shape=(784,)))
```

```
model.add(Activation('relu'))
```

```
model.add(Dropout(0.2))
```

```
model.add(Dense(512))
```

```
model.add(Activation('relu'))
```

```
model.add(Dropout(0.2))
```

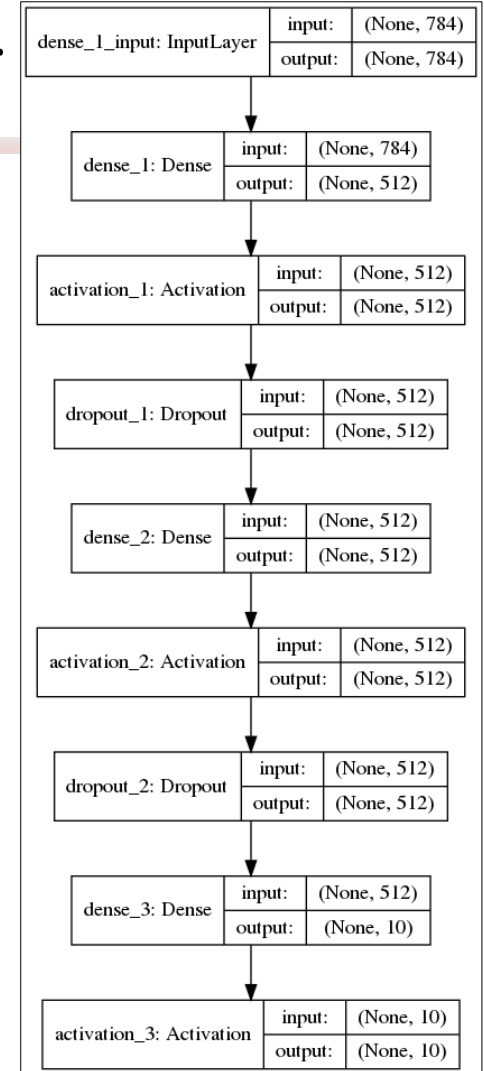
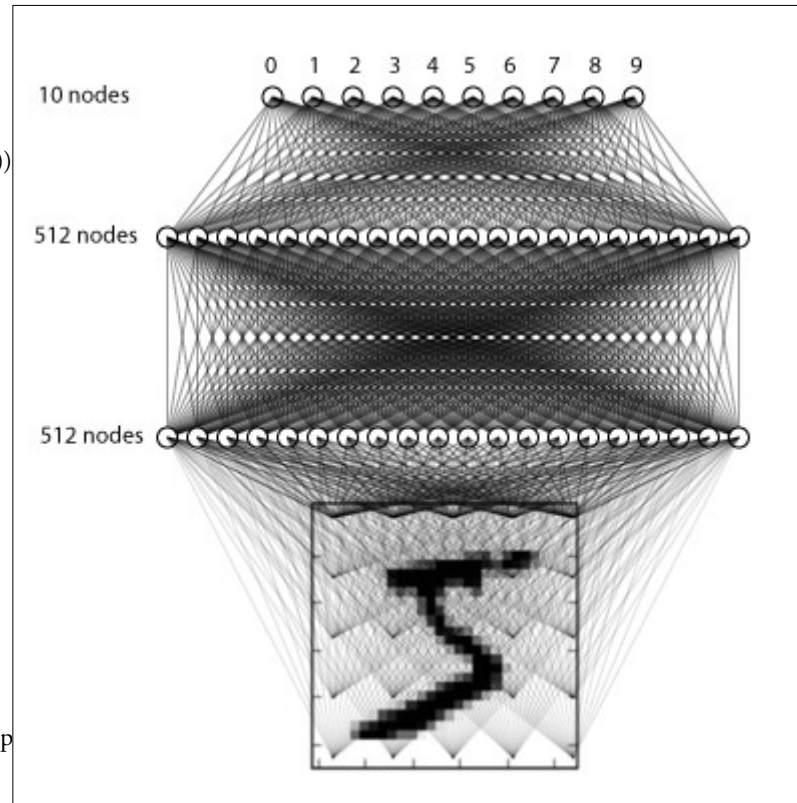
```
model.add(Dense(10))
```

```
model.add(Activation('softmax'))
```

```
# Compile and set loss and optimizer
```

```
model.compile(loss='categorical_crossentropy')
```

```
model.save('PyKerasMNIST.h5')
```



MNISTTMVA.py - Training

Epoch 1/5

59999/59999 [=====] - 5s 84us/step - loss: 0.2476 - accuracy: 0.9253

Epoch 2/5

59999/59999 [=====] - 5s 83us/step - loss: 0.1019 - accuracy: 0.9686

Epoch 3/5

59999/59999 [=====] - 5s 80us/step - loss: 0.0729 - accuracy: 0.9772

Epoch 4/5

59999/59999 [=====] - 5s 77us/step - loss: 0.0565 - accuracy: 0.9819

Epoch 5/5

59999/59999 [=====] - 5s 78us/step - loss: 0.0454 - accuracy: 0.9857

MNISTTMVA.C

// Model generating start

```
TString layoutString("Layout=RELU|512," // First hidden layer with 512 neurons, RELU activation function
                    "RELU|512,"         // Second hidden layer with 512 neurons, RELU activation function
                    "LINEAR|10");       // Final output layer to 10 categories

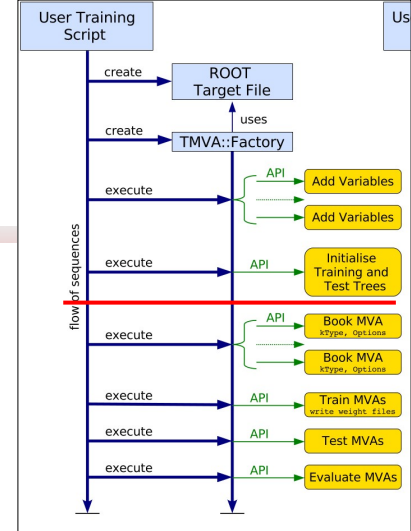
TString training("Repetitions=1,Regularization=None,Multithreading=True,"
                "Optimizer=ADAM,LearningRate=0.001," // Use Adam optimizer with learning rate 0.001
                "MaxEpochs=5,BatchSize=128,"        // Batch size 128 and train 5 times
                "ConvergenceSteps=100,"              // Do not use Convergence check
                "TestRepetitions=1,"                 // Show validation result for every epochs
                "DropConfig=0.2+0.2+0");             // Set dropout. 0.2 for the first and second layers, and 0 for the final layer.
```

```
TString nnOptions("!H:!V:VarTransform=:ErrorStrategy=MUTUALEXCLUSIVE:ValidationSize=128");
```

```
TString trainingStrategyString("TrainingStrategy=");    trainingStrategyString += training;
```

```
nnOptions.Append(":");    nnOptions.Append(layoutString);
```

```
nnOptions.Append(":");    nnOptions.Append(trainingStrategyString);
```



MNISTTMVA.C - Training

: ***** Deep Learning Network *****

DEEP NEURAL NETWORK: Depth = 3 Input = (1, 1, 784) Batch size = 128 Loss function = S

Layer 0 DENSE Layer: (Input = 784 , Width = 512) Output = (1 , 128 , 512) Activation Function = Relu Dropout prob. = 0.8

Layer 1 DENSE Layer: (Input = 512 , Width = 512) Output = (1 , 128 , 512) Activation Function = Relu Dropout prob. = 0.8

Layer 2 DENSE Layer: (Input = 512 , Width = 10) Output = (1 , 128 , 10) Activation Function = Identity

: Using 59872 events for training and 128 for testing

: Training phase 1 of 1: Optimizer ADAM Learning rate = 0.001 regularization 0 minimum error = 2.31272

: -----

: Epoch | Train Err. Val. Err. t(s)/epoch t(s)/Loss nEvents/s Conv. Steps

: -----

: 1 Minimum Test error found - save the configuration

: 1 | **0.106914** 0.0973637 **15.2004** 3.28711 5017.61 0

: 2 Minimum Test error found - save the configuration

: 2 | **0.0731581** 0.0519007 **16.2432** 3.58274 4721.49 0

: 3 | **0.0588579** 0.0568269 **18.624** 4.14863 4129.5 1

: 4 Minimum Test error found - save the configuration

: 4 | **0.0481715** 0.0447528 **17.3988** 3.14962 4195.04 0

: 5 | **0.0418479** 0.0455728 **17.7728** 3.76165 4266.32 1

:

: Elapsed time for training with 60000 events: 85.3 sec

Comparison (DNN) - kPyKeras vs kDL

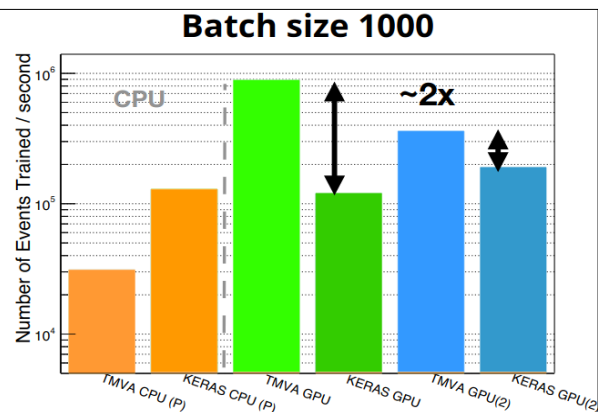
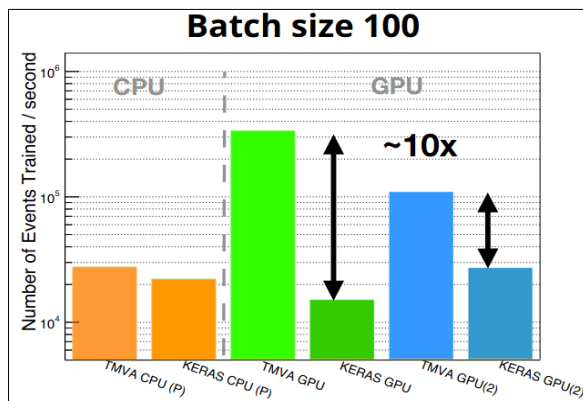
- The accuracy is consistent as they use exactly the same neural network architecture.
 - Can be checked using ‘*Application.C/py’ macros.
- Training time is quite different.
 - kDL : $t / \text{epoch} = 17 \text{ s}$
 - kPyKeras : $t / \text{epoch} = 5 \text{ s}$

Comparison (DNN) - kPyKeras vs kDL

- The accuracy is consistent as they use exactly the same neural network architecture.
 - Can be checked using ‘*Application.C/py’ macros.
- Training time is quite different.
 - kDL : $t / \text{epoch} = 17 \text{ s}$
 - kPyKeras : $t / \text{epoch} = 5 \text{ s}$ —————▶ 3.4 times faster!

Comparison (DNN) - kPyKeras vs kDL

- TMVA group claims that TMVA is faster than Keras for light network.



- Key difference is GPU utilisation
- Tensorflow optimised for large operations

<https://indico.cern.ch/event/587955/contributions/2937501/attachments/1678682/2705216/tmva-chep2018.pdf>

(S) — Single threaded

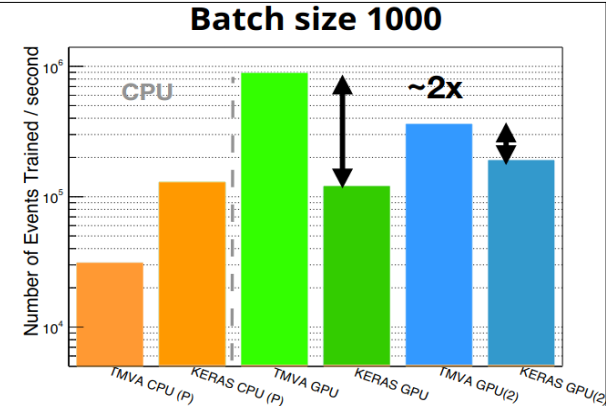
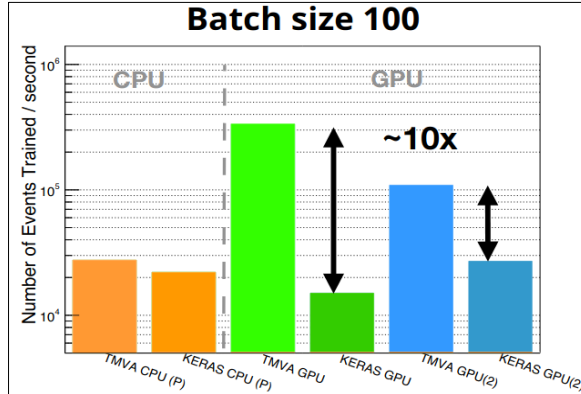
GPU — GTX1080Ti

Comparison (DNN) - kPyKeras vs kDL

- TMVA group claims that TMVA is faster than Keras for light network.
- TMVA seems slower than Keras for heavy net.

<Training time>

kDL : t / epoch = 17 s
kPyKeras : t / epoch = 5 s



- Key difference is GPU utilisation
- Tensorflow optimised for large operations

<https://indico.cern.ch/event/587955/contributions/2937501/attachments/1678682/2705216/tmva-chep2018.pdf>

(S) — Single threaded

GPU — GTX1080Ti

List of Codes

- MNISTtoROOT.py
- MNISTTMVA.py
- MNISTTMVA.C
- MNISTTMVAApplication.C
- MNISTTMVAApplication.py
- MNIST_PyMVA_CNN.py
- MNIST_TMVA_CNN.C

Introduction to CNN

- Convolution applies **kernels (filters)** that traverse through each image and generate **feature maps**.

| | | | | |
|---|-----------------|-----------------|-----------------|---|
| 1 | 1 _{x1} | 1 _{x0} | 0 _{x1} | 0 |
| 0 | 1 _{x0} | 1 _{x1} | 1 _{x0} | 0 |
| 0 | 0 _{x1} | 1 _{x0} | 1 _{x1} | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |

Image

| | | |
|---|---|--|
| 4 | 3 | |
| | | |
| | | |

Convolved
Feature

| | | | | |
|---|---|-----------------|-----------------|-----------------|
| 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 _{x1} | 1 _{x0} | 1 _{x1} |
| 0 | 0 | 1 _{x0} | 1 _{x1} | 0 _{x0} |
| 0 | 1 | 1 _{x1} | 0 _{x0} | 0 _{x1} |

Image

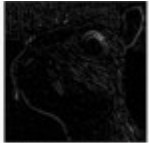

| | | |
|---|---|---|
| 4 | 3 | 4 |
| 2 | 4 | 3 |
| 2 | 3 | 4 |



Convolved
Feature

<https://colab.research.google.com/github/AviatorMoser/keras-mnist-tutorial/blob/master/MNIST%20in%20Keras.ipynb#scrollTo=mBSu1bGzafJ3>

Introduction to CNN

- Convolution was often used in photo-editing software.

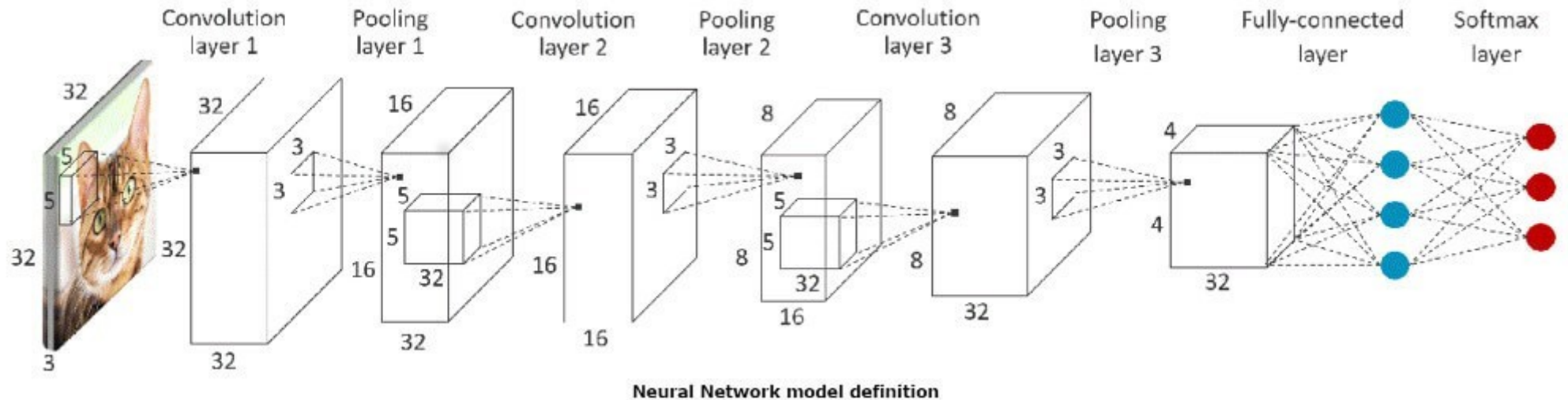
| | | |
|----------------|-----------------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| Edge detection | $\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ |  |
| | $\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$ |  |

| | | |
|----------------------------------|----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| Box blur (normalized) | $\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ |  |
| Gaussian blur (approximation) | $\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$ |  |

<https://colab.research.google.com/github/AviatorMoser/keras-mnist-tutorial/blob/master/MNIST%20in%20Keras.ipynb#scrollTo=mBSu1bGzafJ3>

Introduction to CNN

- Example CNN structure is like this.
 - This is for CIFAR-10. Ours is not this one.



<https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/deploying-convolutional-neural-network-on-cortex-m-with-cmsis-nn>

MNIST_PyMVA_CNN.py

```
model = Sequential()                                # Linear stacking of layers

# Convolution Layer 1
model.add(Reshape((28,28, 1), input_shape=(784,)))
model.add(Conv2D(32, (3, 3))) # 32 different 3x3 kernels -- so 32 feature maps
model.add(BatchNormalization(axis=-1))             # normalize each feature map before activation
convLayer01 = Activation('relu')                   # activation
model.add(convLayer01)

# Convolution Layer 2
model.add(Conv2D(32, (3, 3))) # 32 different 3x3 kernels -- so 32 feature maps
model.add(BatchNormalization(axis=-1))             # normalize each feature map before activation
model.add(Activation('relu'))                      # activation
convLayer02 = MaxPooling2D(pool_size=(2,2))        # Pool the max values over a 2x2 kernel
model.add(convLayer02)

# Convolution Layer 3
model.add(Conv2D(64,(3, 3))) # 64 different 3x3 kernels -- so 64 feature maps
model.add(BatchNormalization(axis=-1))             # normalize each feature map before activation
convLayer03 = Activation('relu')                   # activation
model.add(convLayer03)
```

```
# Convolution Layer 4
model.add(Conv2D(64, (3, 3))) # 64 different 3x3 kernels -- so 64 feature maps
model.add(BatchNormalization(axis=-1))             # normalize each feature map before activation
model.add(Activation('relu')) # activation
convLayer04 = MaxPooling2D(pool_size=(2,2))        # Pool the max values over a 2x2 kernel
model.add(convLayer04)
model.add(Flatten()) # Flatten final 4x4x64 output matrix into a 1024-length vector

# Fully Connected Layer 5
model.add(Dense(512)) # 512 FCN nodes
model.add(BatchNormalization()) # normalization
model.add(Activation('relu')) # activation

# Fully Connected Layer 6
model.add(Dropout(0.2)) # 20% dropout of randomly selected nodes
model.add(Dense(10)) # final 10 FCN nodes
model.add(Activation('softmax')) # softmax activation

# we'll use the same optimizer
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

model.save('MNIST_PyMVA_CNN_Model.h5')
model.summary()
```

MNIST_TMVA_CNN.C

```
/**
```

Reference:

https://github.com/lmoneta/tmva-tutorial/blob/master/notebooks/TMVA_CNN_Classification.C

Book Convolutional Neural Network in TMVA

For building a CNN one needs to define

- Input Layout : number of channels (in this case = 1) | image height | image width
- Batch Layout : batch size | number of channels | image size = (height*width)

Then one add Convolutional layers and MaxPool layers.

- For Convolutional layer the option string has to be:
 - CONV | number of units | filter height | filter width | stride height | stride width | padding height | padding width | activation function
 - note in this case we are using a filter 3x3 and padding=1 and stride=1 so we get the output dimension of the conv layer equal to the input
- For the MaxPool layer:
 - MAXPOOL | pool height | pool width | stride height | stride width

The RESHAPE layer is needed to flatten the output before the Dense layer

Note that to run the CNN is required to have CPU or GPU support

```
***/
```

```
// Generating Model
```

```
TString inputLayoutString("InputLayout=1|28|28");
```

```
TString batchLayoutString("BatchLayout=128|1|784");
```

```
TString convLayer01("CONV|32|3|3|1|1|0|0|RELU");
```

```
TString convLayer02("CONV|32|3|3|1|1|0|0|RELU");
```

```
TString maxPooling02("MAXPOOL|2|2|2|2");
```

```
TString convLayer03("CONV|64|3|3|1|1|0|0|RELU");
```

```
TString convLayer04("CONV|64|3|3|1|1|0|0|RELU");
```

```
TString maxPooling04("MAXPOOL|2|2|2|2");
```

```
TString flatten04("RESHAPE|FLAT");
```

```
TString fullyConnLayer05("DENSE|512|RELU");
```

```
TString fullyConnLayer06("DENSE|10|LINEAR"); // SOFTMAX is included in the loss function.
```

```
TString layoutString("Layout=");
```

```
layoutString += " " + convLayer01;
```

```
layoutString += " " + convLayer02 + " " + maxPooling02;
```

```
layoutString += " " + convLayer03;
```

```
layoutString += " " + convLayer04 + " " + maxPooling04 + " " + flatten04;
```

```
layoutString += " " + fullyConnLayer05;
```

```
layoutString += " " + fullyConnLayer06;
```

Comparison - DNN vs CNN

| Accuracy | DNN | CNN |
|----------|--------|--------|
| Epoch 1 | 92.53% | 97.39% |
| Epoch 2 | 96.86% | 99.16% |
| Epoch 3 | 97.72% | 99.33% |
| Epoch 4 | 98.19% | 99.48% |
| Epoch 5 | 98.57% | 99.64% |
| Test | 98.20% | 99.02% |

Comparison (CNN) - kPyKeras vs kDL

- Training using kDL(C++) takes too long time!
 - kDL : $t / \text{epoch} = 5051 \text{ s} = 1 \text{ h } 14 \text{ min}$
 - kPyKeras : $t / \text{epoch} = 81 \text{ s} = 1 \text{ min}$

Comparison (CNN) - kPyKeras vs kDL

- Training using kDL(C++) takes too long time!
 - kDL : $t / \text{epoch} = 5051 \text{ s} = 1 \text{ h } 14 \text{ min}$
 - kPyKeras : $t / \text{epoch} = 81 \text{ s} = 1 \text{ min}$

—————▶ 62 times faster!

Conclusion

- I've implemented MNIST classifier using DNN / CNN in TMVA(C++) and PyMVA(Python, Keras).
 - All the codes are available in my github.
 - <https://github.com/physmlee/DLStudy>
- TMVA is much slower (3~60times) than PyMVA when using large network.
- As we are going to try large network, I suggest using PyMVA for our project.

Supplementary



MNISTtoROOT.py

```
# Load MNIST Data Set
```

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

Download data.
It is written in .npz format.

```
# Make Tree
```

```
nb_classes = 10
```

```
treelist = []
```

```
for i in range(nb_classes):
```

```
    treelist.append( TTree( 'train%d' %(i), 'MNIST_train%d' %(i) ) )
```

```
    treelist.append( TTree( 'test%d' %(i), 'MNIST_test%d' %(i) ) )
```

MNISTtoROOT.py

```
# Load MNIST Data Set
```

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

```
# Make Tree
```

```
nb_classes = 10
```

```
treelist = []
```

Declare trees.

```
for i in range(nb_classes):
```

```
    treelist.append( TTree( 'train%d' %(i), 'MNIST_train%d' %(i) ) )
```

```
    treelist.append( TTree( 'test%d' %(i), 'MNIST_test%d' %(i) ) )
```

MNISTtoROOT.py

```
# Set Branch
imagelist = []
for tree in treelist:
    for j in range( 28 * 28 ):
        imagelist.append( np.empty( (1), dtype="float32" ) )
        tree.Branch( 'image%d' %(j), imagelist[j], 'image%d/F' %(j) )
```

Link each pixel to one branch.
There will be 784 branches.

```
# Fill Tree
for i in tqdm( range( X_train.shape[0] ), desc='Writing Training Data' ):
    for j in range( 28 * 28 ):
        imagelist[j][0] = X_train[i][j]
    treelist[ 2*y_train[i] ].Fill()

for i in tqdm( range( X_test.shape[0] ), desc='Writing Test Data' ): #
    for j in range( 28 * 28 ):
        imagelist[j][0] = X_test[i][j]
    treelist[ 2*y_test[i] + 1 ].Fill()
```

MNISTtoROOT.py

```
# Set Branch
imagelist = []
for tree in treelist:
    for j in range( 28 * 28 ):
        imagelist.append( np.empty( (1), dtype="float32" ) )
        tree.Branch( 'image%d' %(j), imagelist[j], 'image%d/F' %(j) )
```

```
# Fill Tree
for i in tqdm( range( X_train.shape[0] ), desc='Writing Training Data' ):
    for j in range( 28 * 28 ):
        imagelist[j][0] = X_train[i][j]
    treelist[ 2*y_train[i] ].Fill()
```

```
for i in tqdm( range( X_test.shape[0] ), desc='Writing Test Data' ): #
    for j in range( 28 * 28 ):
        imagelist[j][0] = X_test[i][j]
    treelist[ 2*y_test[i] + 1 ].Fill()
```

Fill in the tree.

MNISTTMVA.py

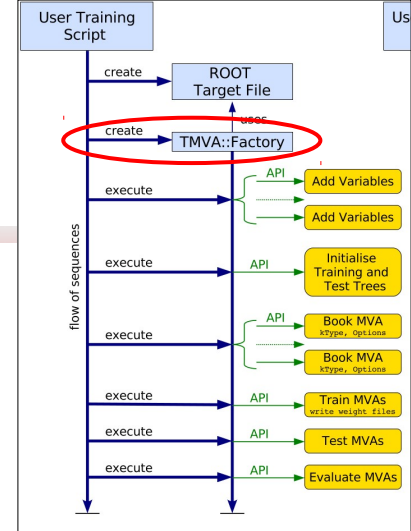
```
# Setup TMVA
```

```
TMVA.Tools.Instance()
```

```
TMVA.PyMethodBase.PyInitialize()
```

```
output = TFile.Open('./data/MNISTTMVA.root', 'RECREATE')
```

```
factory = TMVA.Factory( 'TMVAClassification', output,  
                        '!V:Color:DrawProgressBar:',  
                        '!Silent:',  
                        'Transformations=:',  
                        'AnalysisType=multiclass' )
```



MNISTTMVA.py

```
dataloader = TMVA.DataLoader( 'dataset' )  
for branch in traintree[0].GetListOfBranches():  
    dataloader.AddVariable( branch.GetName() )
```

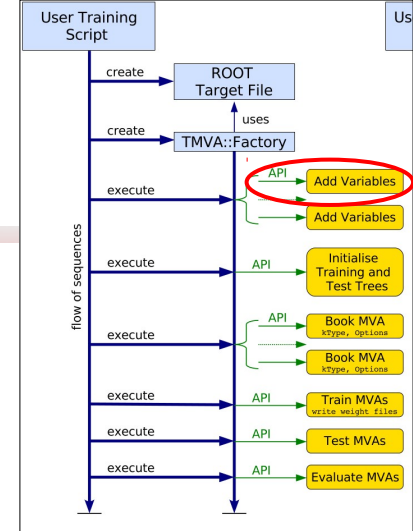
Add variables

```
for i in range( nb_classes ):
```

```
    dataloader.AddTree( traintree[i], '%d' %(i), weight, cut, TMVA.Types.kTraining )
```

```
    dataloader.AddTree( testtree[i] , '%d' %(i), weight, cut, TMVA.Types.kTesting )
```

```
dataloader.PrepareTrainingAndTestTree(cut, '!CalcCorrelations:NormMode=None:!V')
```



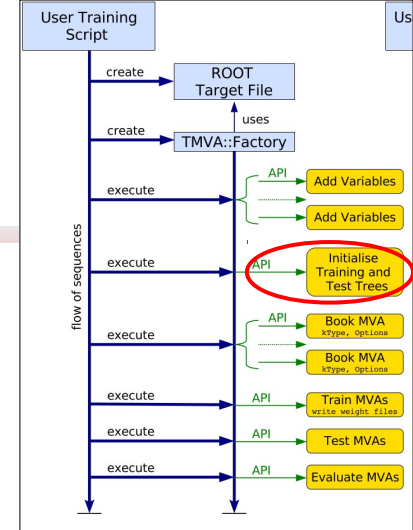
MNISTTMVA.py

```
dataloader = TMVA.DataLoader( 'dataset' )  
for branch in traintree[0].GetListOfBranches():  
    dataloader.AddVariable( branch.GetName() )
```

```
for i in range( nb_classes ):  
    dataloader.AddTree( traintree[i], '%d' %(i), weight, cut, TMVA.Types.kTraining )  
    dataloader.AddTree( testtree[i] , '%d' %(i), weight, cut, TMVA.Types.kTesting )
```

Initialize

```
dataloader.PrepareTrainingAndTestTree(cut, '!CalcCorrelations:NormMode=None:!V')
```



MNISTTMVA.py

```
dataloader = TMVA.DataLoader( 'dataset' )  
for branch in traintree[0].GetListOfBranches():  
    dataloader.AddVariable( branch.GetName() )
```

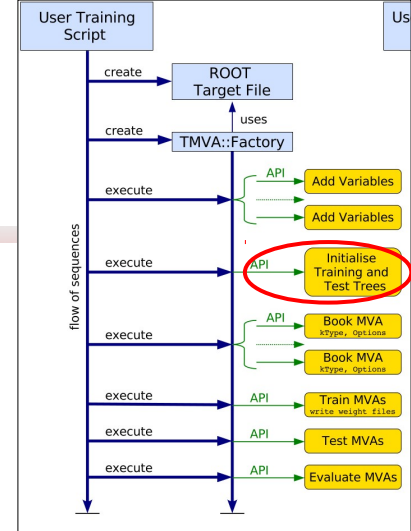
```
for i in range( nb_classes ):
```

```
    dataloader.AddTree( traintree[i], '%d' %(i), weight, cut, TMVA.Types.kTraining )
```

```
    dataloader.AddTree( testtree[i] , '%d' %(i), weight, cut, TMVA.Types.kTesting )
```

It calculates correlation matrix of input variables as default, which is time consuming!

```
dataloader.PrepareTrainingAndTestTree(cut, '!CalcCorrelations:NormMode=None:!V')
```



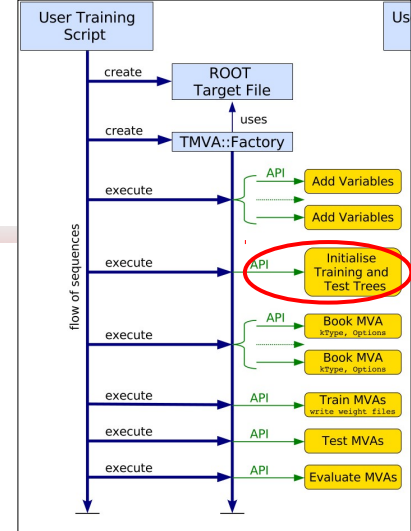
MNISTTMVA.py

```
dataloader = TMVA.DataLoader( 'dataset' )  
  
for branch in traintree[0].GetListOfBranches():  
    dataloader.AddVariable( branch.GetName() )
```

```
for i in range( nb_classes ):  
    dataloader.AddTree( traintree[i], '%d' %(i), weight, cut, TMVA.Types.kTraining )  
    dataloader.AddTree( testtree[i], '%d' %(i), weight, cut, TMVA.Types.kTesting )
```

One can apply some normalization technique to the event numbers of each classes.

```
dataloader.PrepareTrainingAndTestTree(cut, '!CalcCorrelations:NormMode=None:!V')
```



MNISTTMVA.py

```
# Model generating start
```

```
model = Sequential()
```

```
model.add(Dense(512, input_shape=(784,)))
```

```
model.add(Activation('relu'))
```

```
model.add(Dropout(0.2))
```

1st layer. 512 neurons. RELU activation.

```
model.add(Dense(512))
```

```
model.add(Activation('relu'))
```

```
model.add(Dropout(0.2))
```

2nd layer. 512 neurons. RELU activation.

```
model.add(Dense(10))
```

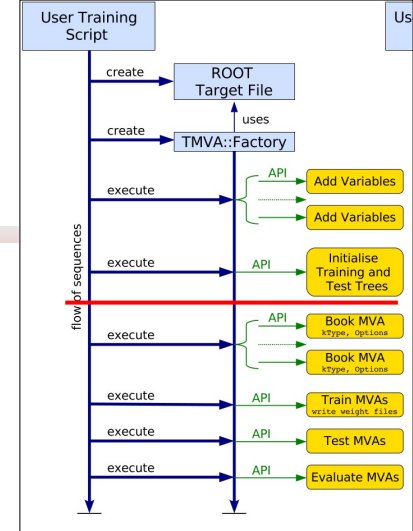
```
model.add(Activation('softmax'))
```

Final layer. 10 neurons. SOFTMAX activation.

```
# Compile and set loss and optimizer
```

```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
model.save('PyKerasMNIST.h5')
```



MNISTTMVA.py

```
# Model generating start
```

```
model = Sequential()
```

```
model.add(Dense(512, input_shape=(784,)))
```

```
model.add(Activation('relu'))
```

```
model.add(Dropout(0.2))
```

```
model.add(Dense(512))
```

```
model.add(Activation('relu'))
```

```
model.add(Dropout(0.2))
```

```
model.add(Dense(10))
```

```
model.add(Activation('softmax'))
```

```
# Compile and set loss as categorical_crossentropy
```

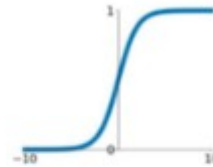
```
model.compile(loss='categorical_crossentropy',
```

```
metrics=['accuracy'], model_path='PyKerasMNIST.h5')
```

Activation Functions

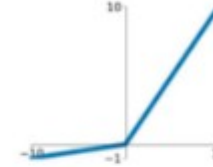
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



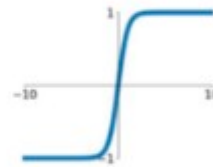
Leaky ReLU

$$\max(0.1x, x)$$



tanh

$$\tanh(x)$$



Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

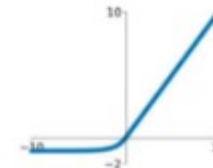
ReLU

$$\max(0, x)$$

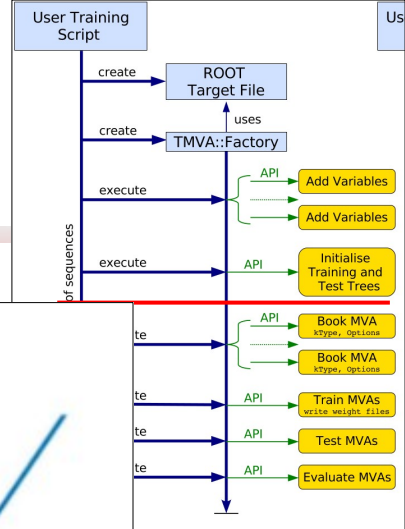


ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Different Activation Functions and their Graphs



MNISTTMVA.py

```
# Model generating start
```

```
model = Sequential()
```

```
model.add(Dense(512, input_shape=(784,)))
```

```
model.add(Activation('relu'))
```

```
model.add(Dropout(0.2))
```

```
model.add(Dense(512))
```

```
model.add(Activation('relu'))
```

```
model.add(Dropout(0.2))
```

```
model.add(Dense(10))
```

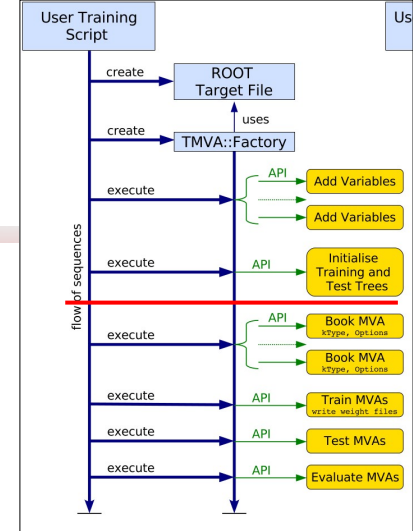
```
model.add(Activation('softmax'))
```

```
# Compile and set loss and optimizer
```

```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
model.save('PyKerasMNIST.h5')
```

Use CROSS ENTROPY loss function and ADAM optimizer.



MNISTTMVA.py

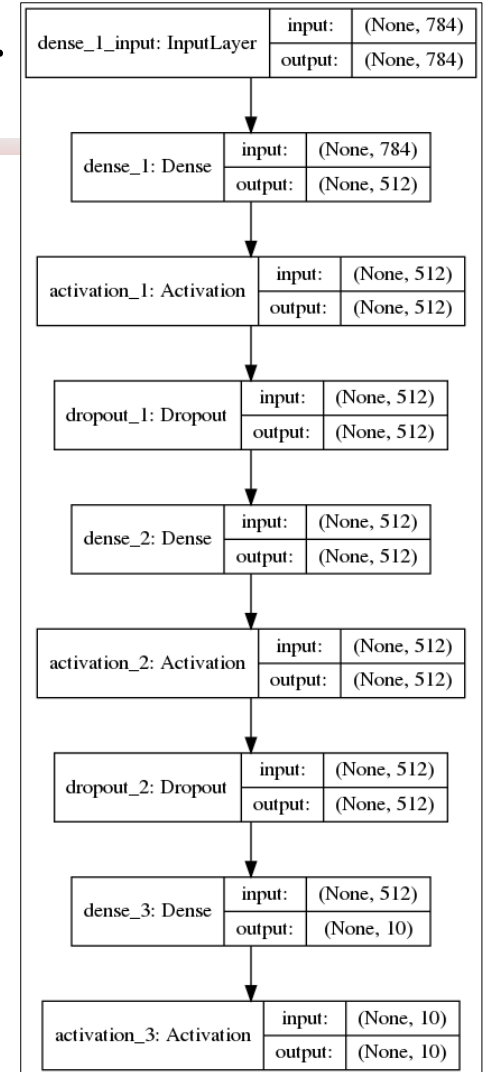
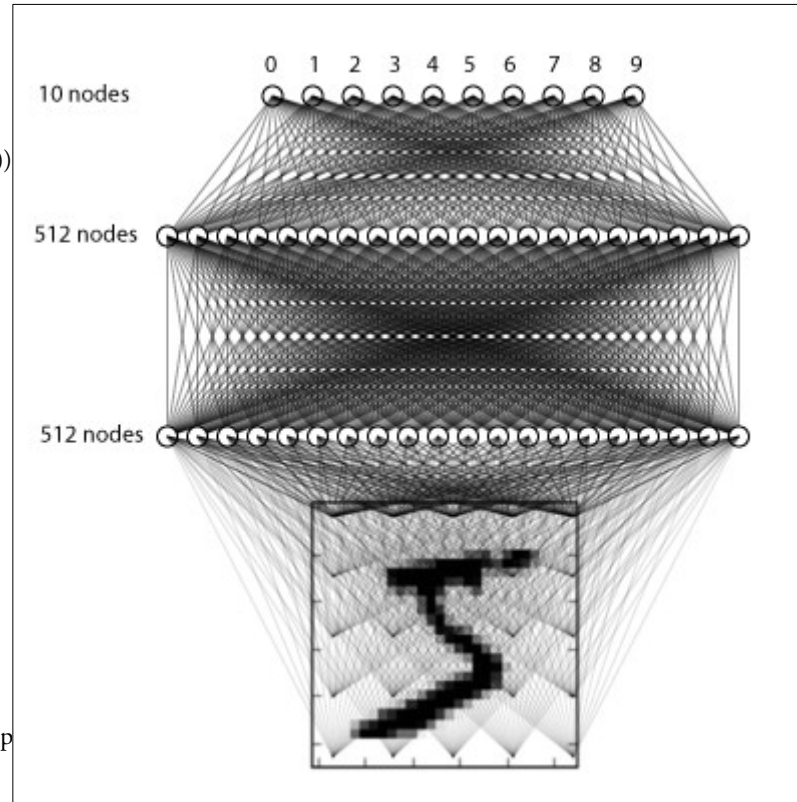
```
# Model generating start
model = Sequential()

model.add(Dense(512, input_shape=(784,)))
model.add(Activation('relu'))
model.add(Dropout(0.2))

model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.2))

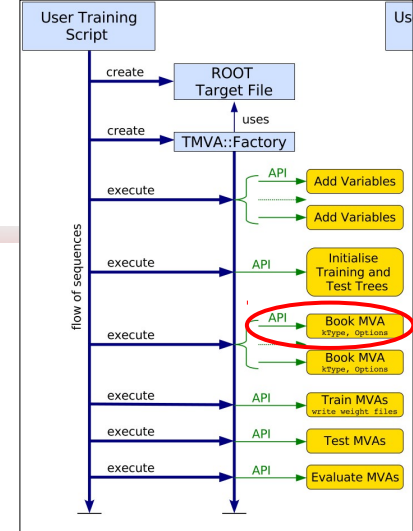
model.add(Dense(10))
model.add(Activation('softmax'))

# Compile and set loss and optimizer
model.compile(loss='categorical_crossentropy')
model.save('PyKerasMNIST.h5')
```



MNISTTMVA.py

```
factory.BookMethod(dataloader, TMVA.Types.kPyKeras,  
    "PyKerasMNIST",  
    '!H:!V:VarTransform=:',  
    'FilenameModel=PyKerasMNIST.h5:',  
    'ValidationSize=1:' # At least one data must be given to validation dataset.  
    '!SaveBestOnly:' # Save the last result, not the best one.  
    'NumEpochs=5:' # Train 5 times  
    'BatchSize=128') # Calculate gradient descent using 128 samples
```

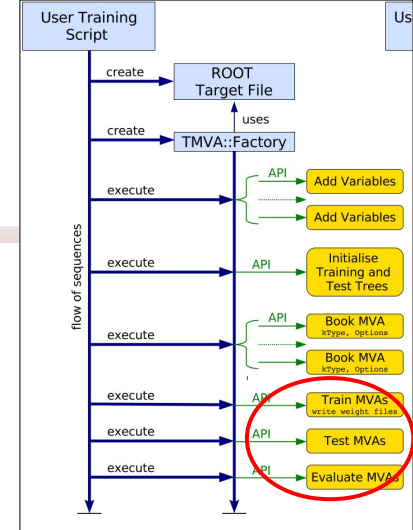


MNISTTMVA.py

factory.TrainAllMethods()

factory.TestAllMethods()

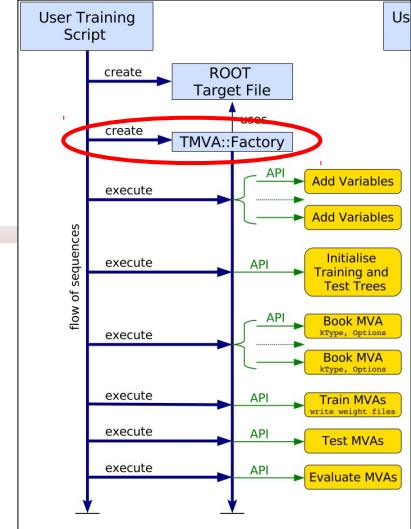
factory.EvaluateAllMethods()



MNISTTMVA.C

```
TMVA::Tools::Instance();  
TString outfileName = "./data/MNISTTMVA.root";  
TFile *output = TFile::Open(outfileName, "RECREATE");
```

```
TMVA::Factory *factory = new TMVA::Factory("TMVAMulticlass", output,  
    "!V:Color:DrawProgressBar:!Silent:"  
    "Transformations=:" // No preprocessing for input variable  
    "AnalysisType=multiclass"); // It is a multiclass classification example
```



MNISTTMVA.C

```
TMVA::DataLoader *dataloader = new TMVA::DataLoader("dataset");
```

```
for (Int_t i = 0; i < pixel; i++) {
```

```
    sprintf(branchname, "image%d", i);
```

```
    dataloader->AddVariable(branchname, 'F'); // register all the 784 pixels as variable
```

```
}
```

```
for (Int_t i = 0; i < nb_classes; i++) {
```

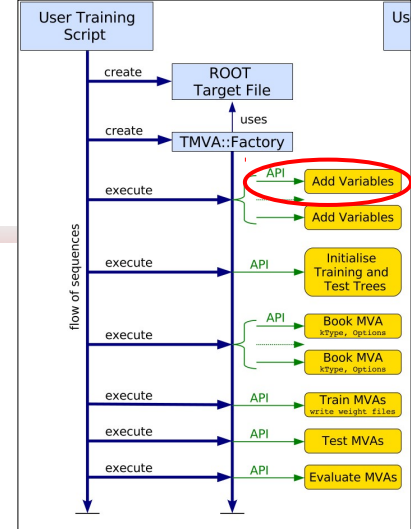
```
    sprintf(classname, "%d", i);
```

```
    dataloader->AddTree(traintree[i], classname, weight, cut, TMVA::Types::kTraining); // Add trees specifying their purpose (Training)
```

```
    dataloader->AddTree(testtree[i], classname, weight, cut, TMVA::Types::kTesting); // Add trees specifying their purpose (Testing)
```

```
}
```

```
dataloader->PrepareTrainingAndTestTree(cut, "!CalcCorrelations:NormMode=None!V");
```



MNISTTMVA.C

```
TMVA::DataLoader *dataloader = new TMVA::DataLoader("dataset");
```

```
for (Int_t i = 0; i < pixel; i++) {
```

```
    sprintf(branchname, "image%d", i);
```

```
    dataloader->AddVariable(branchname, 'F'); // register all the 784 pixels as variable
```

```
}
```

```
for (Int_t i = 0; i < nb_classes; i++) {
```

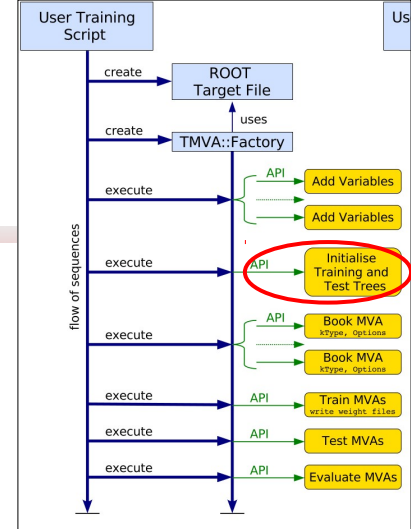
```
    sprintf(classname, "%d", i);
```

```
    dataloader->AddTree(traintree[i], classname, weight, cut, TMVA::Types::kTraining); // Add trees specifying their purpose (Training)
```

```
    dataloader->AddTree(testtree[i], classname, weight, cut, TMVA::Types::kTesting); // Add trees specifying their purpose (Testing)
```

```
}
```

```
dataloader->PrepareTrainingAndTestTree(cut, "!CalcCorrelations:NormMode=None!V");
```



MNISTTMVA.C

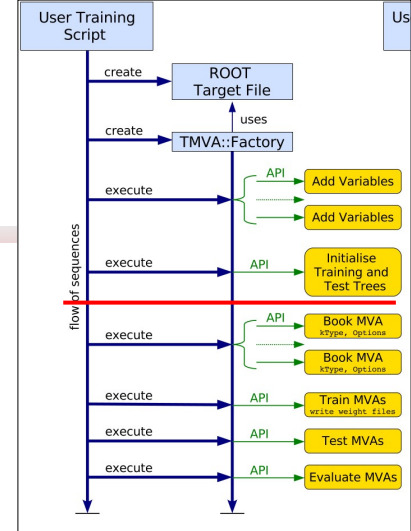
// Model generating start

```
TString layoutString("Layout=RELU|512," // First hidden layer with 512 neurons, RELU activation function
                    "RELU|512,"         // Second hidden layer with 512 neurons, RELU activation function
                    "LINEAR|10");        // Final output layer to 10 categories

TString training("Repetitions=1,Regularization=None,Multithreading=True,"
                "Optimizer=ADAM,LearningRate=0.001," // Use Adam optimizer with learning rate 0.001
                "MaxEpochs=5,BatchSize=128,"        // Batch size 128 and train 5 times
                "ConvergenceSteps=100,"              // Do not use Convergence check
                "TestRepetitions=1,"                 // Show validation result for every epochs
                "DropConfig=0.2+0.2+0");             // Set dropout. 0.2 for the first and second layers, and 0 for the final layer.

TString nnOptions("!H:!V:VarTransform=:ErrorStrategy=MUTUALEXCLUSIVE:ValidationSize=128");

TString trainingStrategyString("TrainingStrategy=");    trainingStrategyString += training;
nnOptions.Append(":");    nnOptions.Append(layoutString);
nnOptions.Append(":");    nnOptions.Append(trainingStrategyString);
```



MNISTTMVA.C

// Model generating start

```
TString layoutString("Layout=RELU|512," // First hidden layer with 512 neurons, RELU activation function
                    "RELU|512,"         // Second hidden layer with 512 neurons, RELU activation function
                    "LINEAR|10");       // Final output layer to 10 categories

TString training("Repetitions=1,Regularization=None,Multithreading=True,"
                "Optimizer=ADAM,LearningRate=0.001," // Use Adam optimizer with learning rate 0.001
                "MaxEpochs=5,BatchSize=128,"        // Batch size 128 and train 5 times
                "ConvergenceSteps=100,"              // Do not use Convergence check
                "TestRepetitions=1,"                 // Show validation result for every epochs
                "DropConfig=0.2+0.2+0");             // Set dropout. 0.2 for the first and second layers, and 0 for the final layer.
```

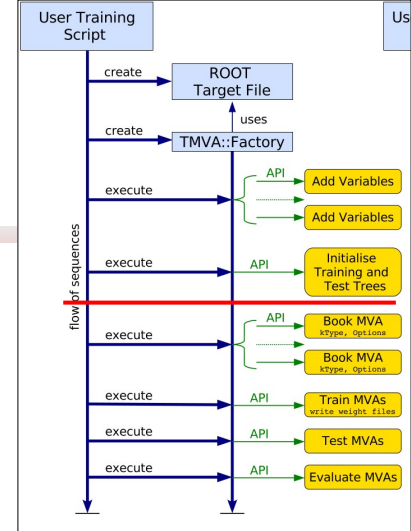
```
TString nnOptions("!H:!V:VarTransform=:ErrorStrategy=MUTUALEXCLUSIVE:ValidationSize=128");
```

MUTUALEXCLUSIVE=SOFTMAX * CROSS ENTROPY

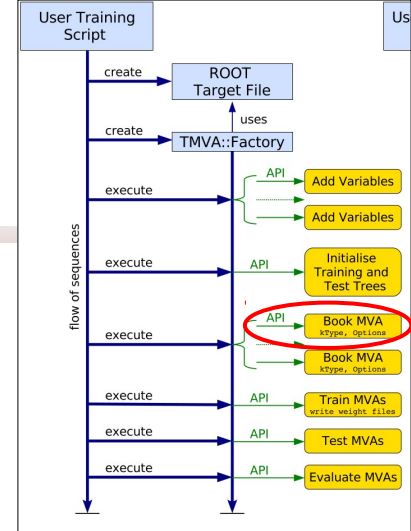
```
TString trainingStrategyString("TrainingStrategy="); trainingStrategyString += training;
```

```
nnOptions.Append(":"); nnOptions.Append(layoutString);
```

```
nnOptions.Append(":"); nnOptions.Append(trainingStrategyString);
```



MNISTTMVA.C



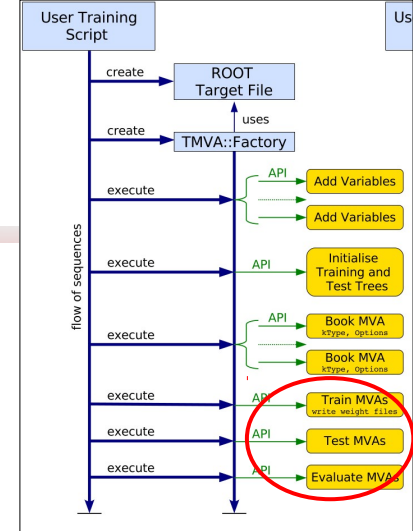
factory->BookMethod(dataloader, TMVA::Types::kDL, "MNISTTMVA", nnOptions);

factory->TrainAllMethods();

factory->TestAllMethods();

factory->EvaluateAllMethods();

MNISTTMVA.C



```
factory->BookMethod(dataloader, TMVA::Types::kDL, "MNISTTMVA", nnOptions);
```

```
factory->TrainAllMethods();
```

```
factory->TestAllMethods();
```

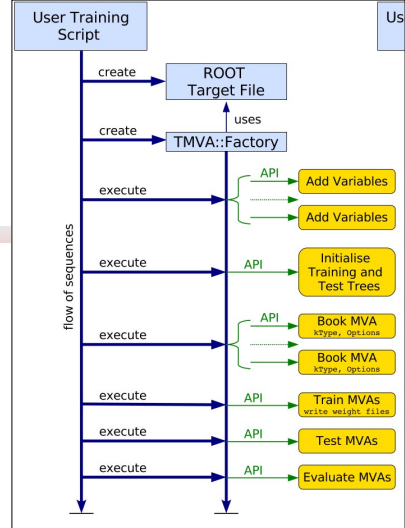
```
factory->EvaluateAllMethods();
```

MNISTTMVA.C

We should use kDL method. It has much various functions than kDNN. Unfortunately, description for kDL is not in the UsersGuide.

We should refer to its source code in the link below;

https://root.cern.ch/doc/master/MethodDL_8cxx_source.html#l00161



```
factory->BookMethod(dataloader, TMVA::Types::kDL, "MNISTTMVA", nnOptions);
```

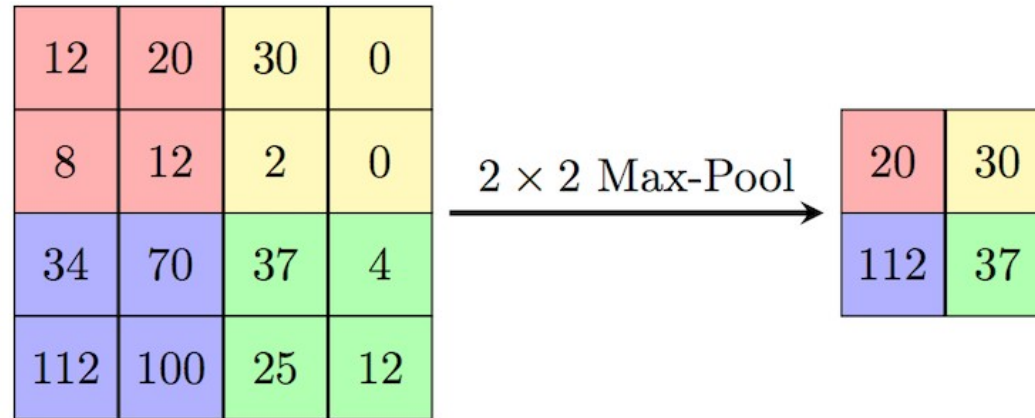
```
factory->TrainAllMethods();
```

```
factory->TestAllMethods();
```

```
factory->EvaluateAllMethods();
```


Introduction to CNN

- Max Pooling is applied to discard non-highlighted elements and to cut down on the computational cost.



<https://colab.research.google.com/github/AviatorMoser/keras-mnist-tutorial/blob/master/MNIST%20in%20Keras.ipynb#scrollTo=mBSu1bGzafJ3>

MNIST_TMVA_CNN.C

```
/**
```

Reference:

https://github.com/lmoneta/tmva-tutorial/blob/master/notebooks/TMVA_CNN_Classification.C

Book Convolutional Neural Network in TMVA

For building a CNN one needs to define

- Input Layout : number of channels (in this case = 1) | image height | image width
- Batch Layout : batch size | number of channels | image size = (height*width)

Then one add Convolutional layers and MaxPool layers.

- For Convolutional layer the option string has to be:
 - CONV | number of units | filter height | filter width | stride height | stride width | padding height | padding width | activation function
 - note in this case we are using a filter 3x3 and padding=1 and stride=1 so we get the output dimension of the conv layer equal to the input
- For the MaxPool layer:
 - MAXPOOL | pool height | pool width | stride height | stride width

The RESHAPE layer is needed to flatten the output before the Dense layer

Note that to run the CNN is required to have CPU or GPU support

```
***/
```

```
// Generating Model
```

```
TString inputLayoutString("InputLayout=1|28|28");
```

```
TString batchLayoutString("BatchLayout=128|1|784");
```

```
TString convLayer01("CONV|32|3|3|1|1|0|0|RELU");
```

TS Convolutional network having 32
TS 3*3 filters, moving (1, 1), without
TS padding, using activation function
TS RELU.

```
TString flatten04("RESHAPE|FLAT");
```

```
TString fullyConnLayer05("DENSE|512|RELU");
```

```
TString fullyConnLayer06("DENSE|10|LINEAR"); // SOFTMAX is included in the loss function.
```

```
TString layoutString("Layout=");
```

```
layoutString += " " + convLayer01;
```

```
layoutString += " " + convLayer02 + " " + maxPooling02;
```

```
layoutString += " " + convLayer03;
```

```
layoutString += " " + convLayer04 + " " + maxPooling04 + " " + flatten04;
```

```
layoutString += " " + fullyConnLayer05;
```

```
layoutString += " " + fullyConnLayer06;
```