KTP Machine Learning Coding Test

Introduction:

Aim: To develop a prototype model using Python for human action recognition from video.
Core Objectives:
1. To develop a 2D CNN model for human action recognition.
2. To develop a 3D CNN model for human action recognition.

Advance Objectives:
3. To develop a 3D CNN with Optical Flow model (concatenate the features extracted from CNN and Optical Flow) for human action recognition.

Methodology & Implementation:

Data Collection:

For this project, I have used a preexisting video data repository (KTH data set). The KTH video database contains six types of human actions (walking, jogging, running, boxing, hand waving and hand clapping) performed several times by 25 subjects in four different scenarios: outdoors *s1*, outdoors with scale variation *s2*, outdoors with different clothes *s3* and indoors *s4* as illustrated below. Currently, the database contains 2391 sequences. All sequences were taken over homogeneous backgrounds with a static camera with a *25*fps frame rate. The sequences were down sampled to the spatial resolution of *160x120* pixels and have a length of four seconds on average. There are about 350 – 600 frames per video.
All sequences are stored using AVI file format and are available online (DIVX-compressed version). There are *25x6x4=600* video files for each combination of 25 subjects, 6 actions and 4 scenarios.

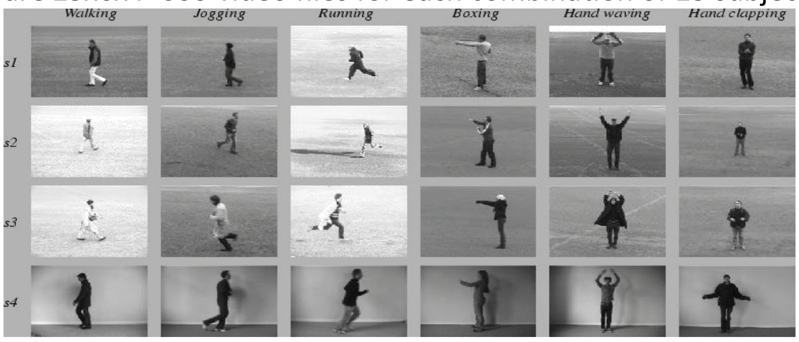
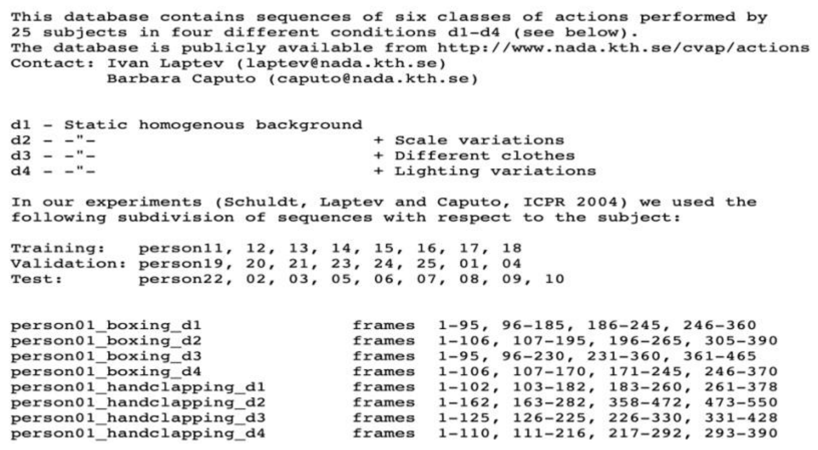


Figure 1 Sample Dataset

Pre-processing:

In this project, all sequences were divided with respect to the subjects into a training set (8 persons), a validation set (8 persons) and a test set (9 persons). The classifiers were trained on a training set while the validation set was used to optimize the parameters of each method. The recognition results were obtained on the test set.
Each file contains about four subsequence's used as a *sequence* in our experiments. The subdivision of each file into sequences in terms *start_frame* and *end_frame* as well as the list of all sequences is given in 00sequences.txt



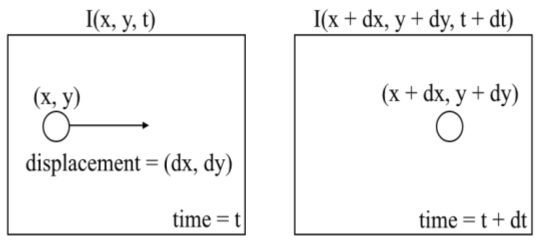
```
This database contains sequences of six classes of actions performed by
25 subjects in four different conditions d1-d4 (see below).
The database is publicly available from http://www.nada.kth.se/cvap/actions
Contact: Ivan Laptev (laptev@nada.kth.se)
         Barbara Caputo (caputo@nada.kth.se)


d1 - Static homogenous background
d2 - -"-                         + Scale variations
d3 - -"-                         + Different clothes
d4 - -"-                         + Lighting variations

In our experiments (Schuldt, Laptev and Caputo, ICPR 2004) we used the
following subdivision of sequences with respect to the subject:

Training:   person11, 12, 13, 14, 15, 16, 17, 18
Validation: person19, 20, 21, 23, 24, 25, 01, 04
Test:       person22, 02, 03, 05, 06, 07, 08, 09, 10


person01_boxing_d1              frames  1-95, 96-185, 186-245, 246-360
person01_boxing_d2              frames  1-106, 107-195, 196-265, 305-390
person01_boxing_d3              frames  1-95, 96-230, 231-360, 361-465
person01_boxing_d4              frames  1-106, 107-170, 171-245, 246-370
person01_handclapping_d1        frames  1-102, 103-182, 183-260, 261-378
person01_handclapping_d2        frames  1-162, 163-282, 358-472, 473-550
person01_handclapping_d3        frames  1-125, 126-225, 226-330, 331-428
person01_handclapping_d4        frames  1-110, 111-216, 217-292, 293-390
```

Figure 2 Data structure and instructions

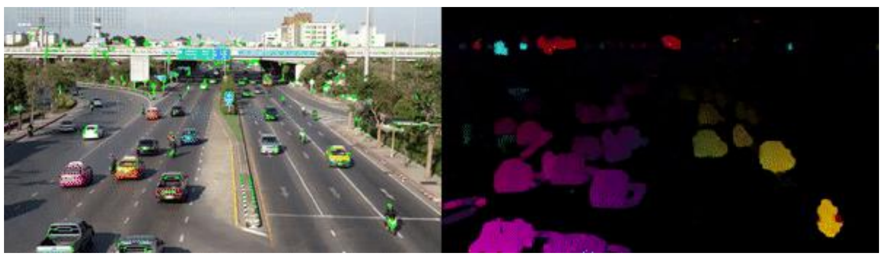
In the first stage of pre-processing, I have extracted and sorted the video data according to the 00sequence.txt instruction file. Then the pre-processed data is stored in 'train.p', 'dev.p' (validation data) and 'train.p'(Which are python pickle binary files) and these data sets consists of the binary format of each frame of video and the video categories. The 'read_dataset()' from 'class RawDataset(Dataset)' returns the instances(video-frame binary data) and their categories to train.py

Optical Flow:

Optical flow is the motion of objects between consecutive frames of sequence, caused by the relative movement between the object and camera.
The first image I(x,y,t) and move its pixels by (dx,dy) over t time, we obtain the new image I(x+dx,y+dy,t+dt).



There are two types of optical flow algorithms
Sparse optical flow:
Sparse optical flow gives the flow vectors of some "interesting features" (say a few pixels depicting the edges or corners of an object) within the frame
Dense optical flow:
Dense Optical flow computes the optical flow vector for every pixel of the frame which may be responsible for its slow speed but leads to a better accurate result.



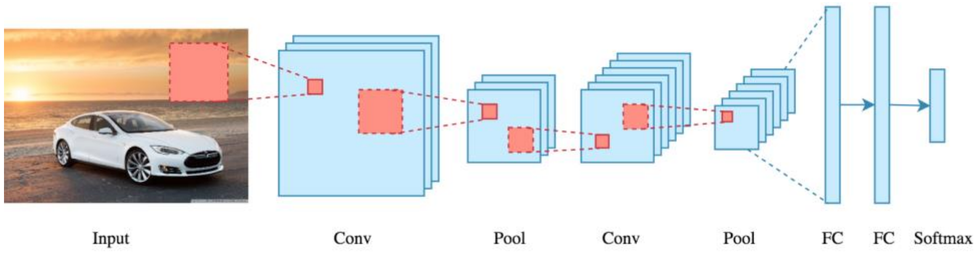
Left: Sparse Optical Flow – track few "features" pixels
Right: Dense Optical Flow – Estimate flow of all pixels in the image

Deep Learning Model:

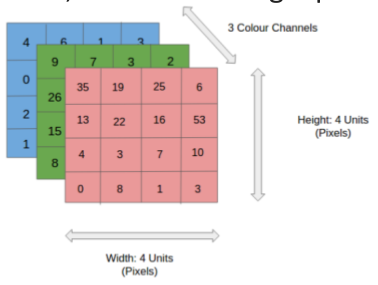
In the past few decades, Deep Learning has proved to be a very powerful tool because of its ability to handle large amounts of data. The interest to use hidden layers has surpassed traditional techniques, especially in pattern recognition. One of the most popular deep neural networks is the Convolutional Neural network.

2D Convolutional Neural Network (CNN):

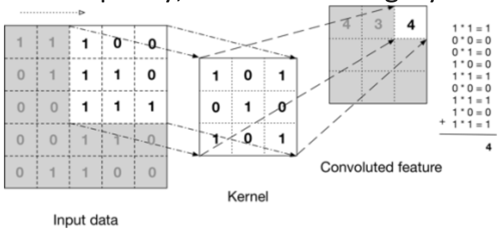
In deep learning, a **convolutional neural network** (**CNN/ConvNet**) is a class of deep neural networks, most commonly applied to analyse visual imagery. Now when we think of a neural network, we think about matrix multiplications but that is not the case with ConvNet. It uses a special technique called Convolution. Now in mathematics **convolution** is a mathematical operation on two functions that produces a third function that expresses how the shape of one is modified by the other.



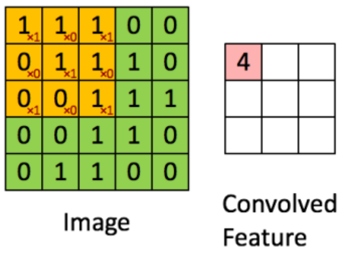
An RGB image is nothing but a matrix of pixel values having three planes whereas a grayscale image is the same, but it has a single plane. Look at this image to understand more.



For simplicity, let's stick with grayscale images as we try to understand how CNNs work.

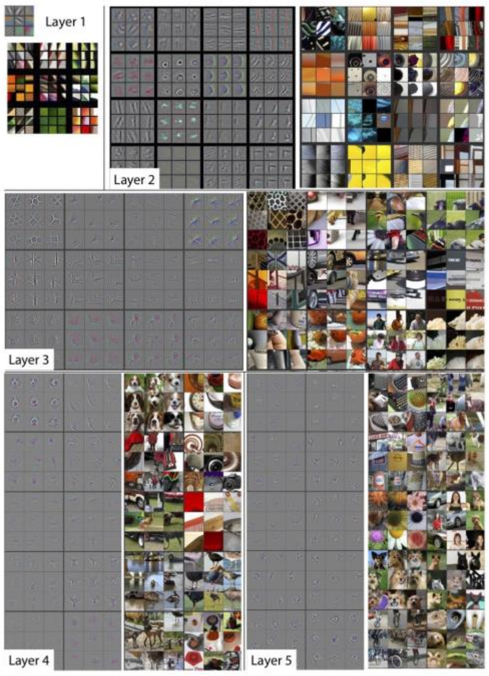


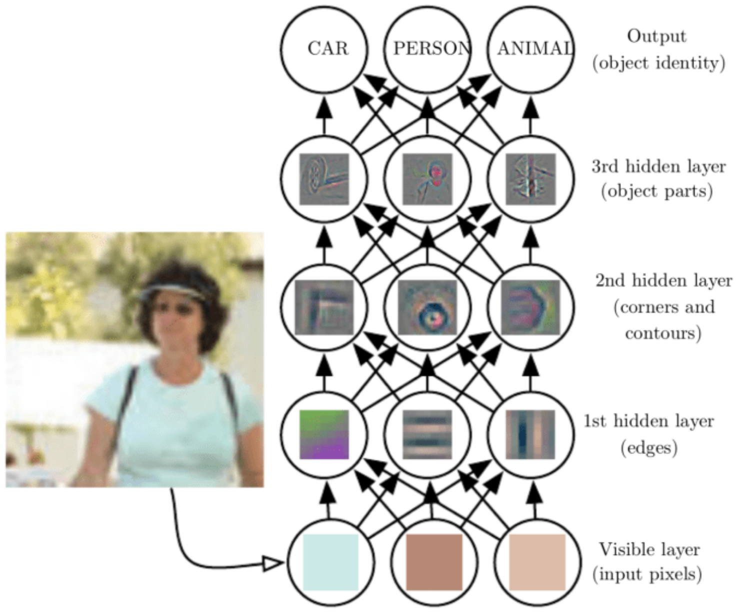
The above image shows what a convolution is. We take a filter/kernel (3×3 matrix) and apply it to the input image to get the convolved feature. This convolved feature is passed on to the next layer.



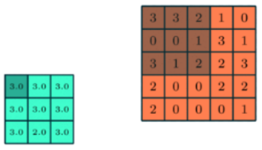
Convolutional neural networks are composed of multiple layers of artificial neurons. Artificial neurons, a rough imitation of their biological counterparts, are mathematical functions that calculate the weighted sum of multiple inputs and outputs an activation value. When you input an image in a ConvNet, each layer generates several activation functions that are passed on to the next layer.

The first layer usually extracts basic features such as horizontal or diagonal edges. This output is passed on to the next layer which detects more complex features such as corners or combinational edges. As we move deeper into the network it can identify even more complex features such as objects, faces, etc.

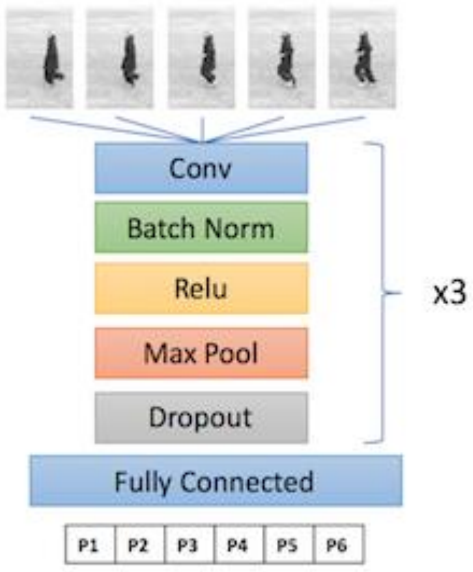


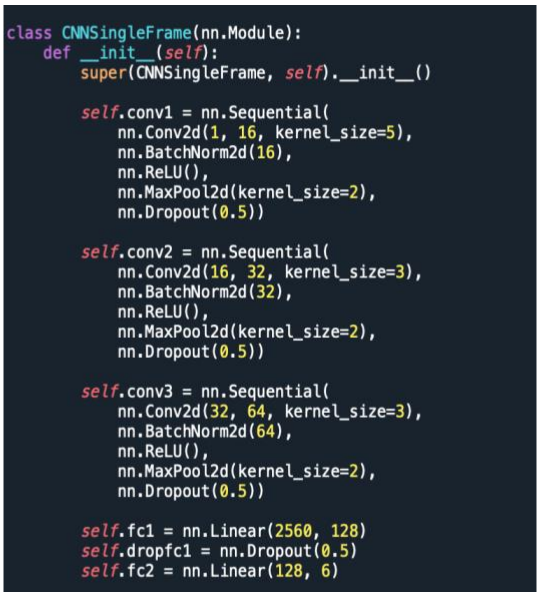


Based on the activation map of the final convolution layer, the classification layer outputs a set of confidence scores (values between 0 and 1) that specify how likely the image is to belong to a "class." For instance, if you have a ConvNet that detects cats, dogs, and horses, the output of the final layer is the possibility that the input image contains any of those animals.
Similar to the Convolutional Layer, the Pooling layer is responsible for reducing the spatial size of the Convolved Feature. This is to decrease the computational power required to process the data by reducing the dimensions. There are two types of pooling average pooling and max pooling.



So, what we do in Max Pooling is find the maximum value of a pixel from a portion of the image covered by the kernel. Max Pooling also performs as a Noise Suppressant. It discards the noisy activations altogether and performs de-noising along with dimensionality reduction. The 2d CNN is trained by sending individual frames as an input

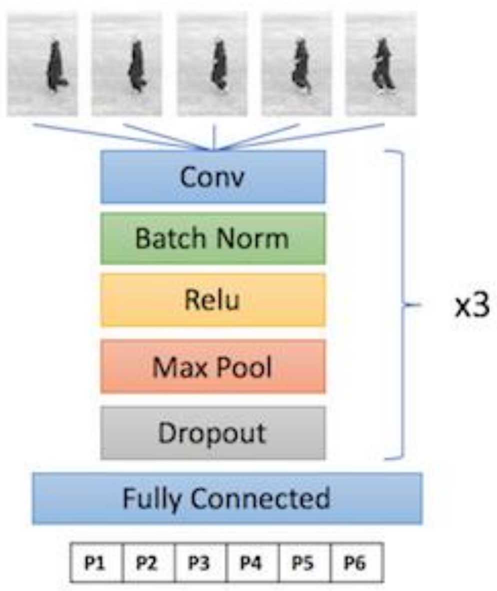


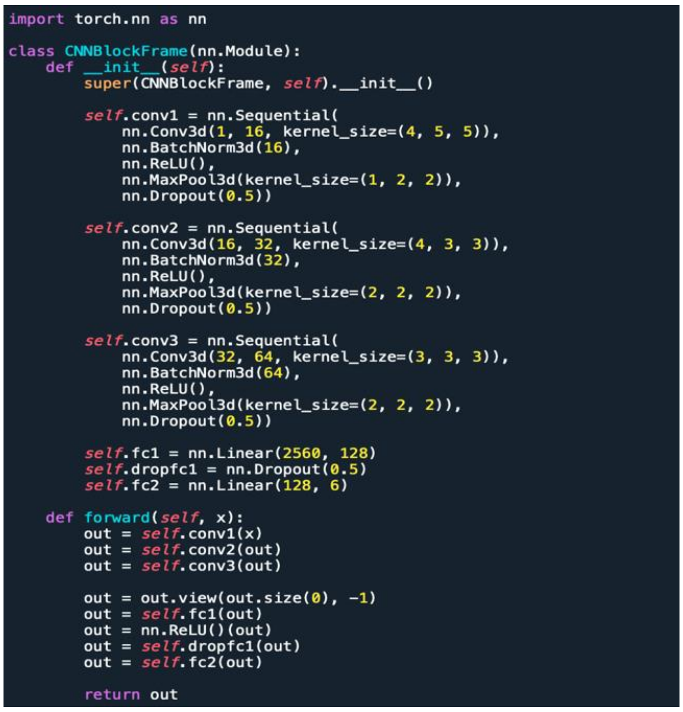


```
class CNNSingleFrame(nn.Module):
    def __init__(self):
        super(CNNSingleFrame, self).__init__()

        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=5),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
            nn.Dropout(0.5))

        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=3),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
            nn.Dropout(0.5))

        self.conv3 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=3),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
            nn.Dropout(0.5))

        self.fc1 = nn.Linear(2560, 128)
        self.dropfc1 = nn.Dropout(0.5)
        self.fc2 = nn.Linear(128, 6)
```

3D Convolutional Neural Network:

A 3d CNN remains regardless of what we say a CNN that is very much similar to a 2d CNN. Except that it differs in these following points (non-exhaustive listing):
3d Convolution Layers
3d MaxPool Layers
For each video, we divide it into blocks of 'n' contiguous frames. The model is then trained on these blocks instead of individual frames. In the convolutional layers, we use 3D convolutional filters to train the model to learn to detect temporal features.

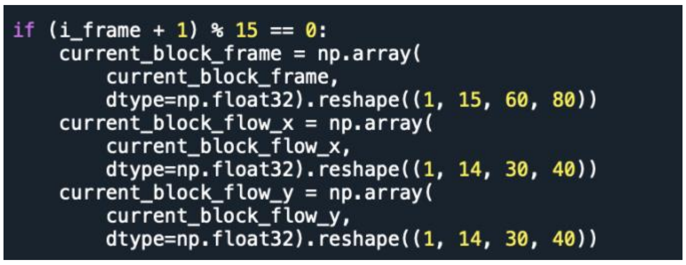




```python
import torch.nn as nn

class CNNBlockFrame(nn.Module):
    def __init__(self):
        super(CNNBlockFrame, self).__init__()

        self.conv1 = nn.Sequential(
            nn.Conv3d(1, 16, kernel_size=(4, 5, 5)),
            nn.BatchNorm3d(16),
            nn.ReLU(),
            nn.MaxPool3d(kernel_size=(1, 2, 2)),
            nn.Dropout(0.5))

        self.conv2 = nn.Sequential(
            nn.Conv3d(16, 32, kernel_size=(4, 3, 3)),
            nn.BatchNorm3d(32),
            nn.ReLU(),
            nn.MaxPool3d(kernel_size=(2, 2, 2)),
            nn.Dropout(0.5))

        self.conv3 = nn.Sequential(
            nn.Conv3d(32, 64, kernel_size=(3, 3, 3)),
            nn.BatchNorm3d(64),
            nn.ReLU(),
            nn.MaxPool3d(kernel_size=(2, 2, 2)),
            nn.Dropout(0.5))

        self.fc1 = nn.Linear(2560, 128)
        self.dropfc1 = nn.Dropout(0.5)
        self.fc2 = nn.Linear(128, 6)

    def forward(self, x):
        out = self.conv1(x)
        out = self.conv2(out)
        out = self.conv3(out)

        out = out.view(out.size(0), -1)
        out = self.fc1(out)
        out = nn.ReLU()(out)
        out = self.dropfc1(out)
        out = self.fc2(out)

        return out
```

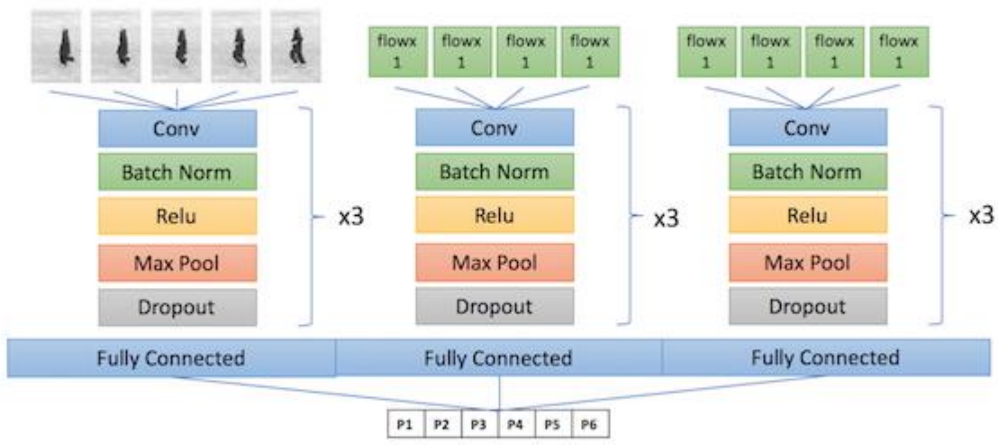
In this project, 15 contiguous frames were used to train the data.3D CNN with Optical Flow:

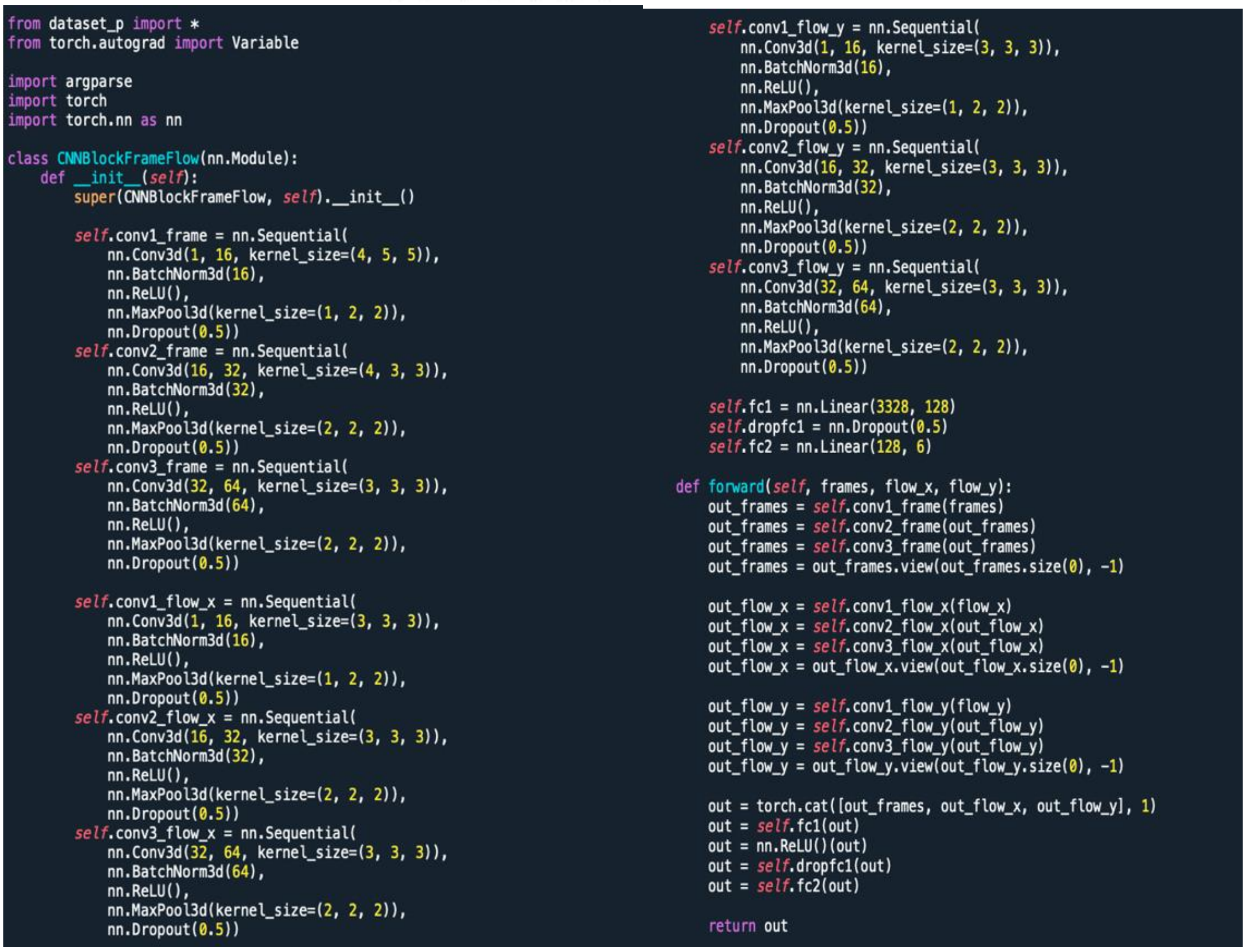
This model is the same as 3D CNN, but additionally, the model needs to be trained on optical flow features. In order to recognize the human action/the displacement of all pixels between two consecutive frames, we are using the dense optical flow Franeback Method.
From the raw data first, I have produced block frames by joining 15 frames for each batch. Then to generate a train, valid and test dataset for 3d CNN with optical flow, taking optical flow data which was already produced in preprocessing step and append this data to each block of 15 frames with both optical flow x-axis and optical flow y-axis.



```python
if (i_frame + 1) % 15 == 0:
    current_block_frame = np.array(
        current_block_frame,
        dtype=np.float32).reshape((1, 15, 60, 80))
    current_block_flow_x = np.array(
        current_block_flow_x,
        dtype=np.float32).reshape((1, 14, 30, 40))
    current_block_flow_y = np.array(
        current_block_flow_y,
        dtype=np.float32).reshape((1, 14, 30, 40))
```

The fully connected layer is a single layer with multiple partitions. For example: In the model below CNNBlockFrameFlow, There are 3 convolutional layers('conv1,2,3_frames') and 3 optical flow x-axis layers ('conv1,2,3_flow_x') and 3 optical flow y-axis layers ('conv1,2,3_flow_y'). Each conv_frame layer is concatenated with their respective conv_flow_x and conv_flow_y layers.

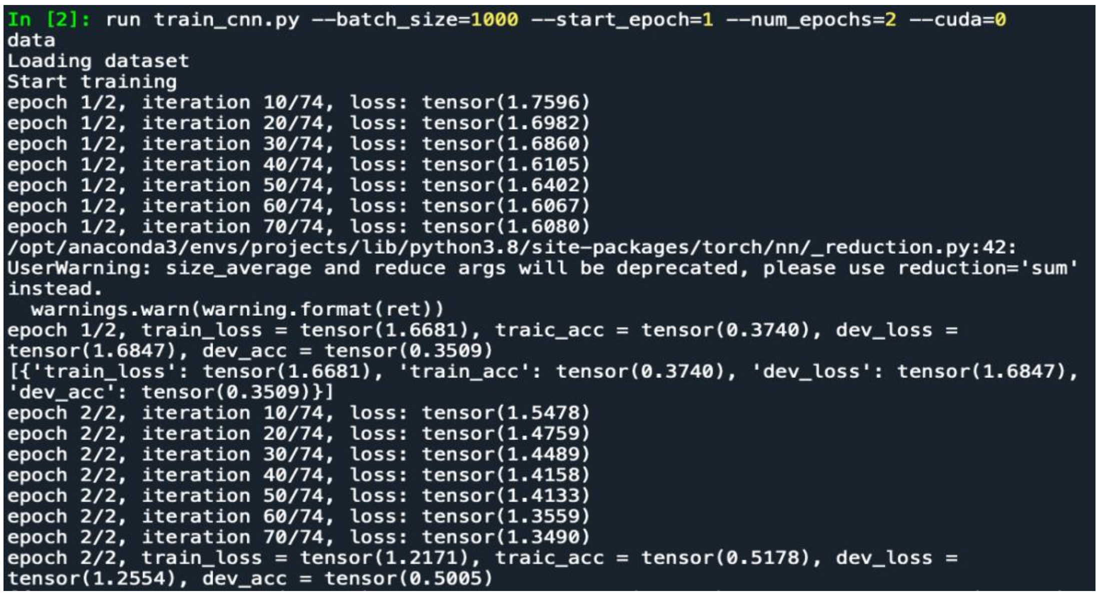




```python
from dataset_p import *
from torch.autograd import Variable

import argparse
import torch
import torch.nn as nn

class CNNBlockFrameFlow(nn.Module):
    def __init__(self):
        super(CNNBlockFrameFlow, self).__init__()

        self.conv1_frame = nn.Sequential(
            nn.Conv3d(1, 16, kernel_size=(4, 5, 5)),
            nn.BatchNorm3d(16),
            nn.ReLU(),
            nn.MaxPool3d(kernel_size=(1, 2, 2)),
            nn.Dropout(0.5))
        self.conv2_frame = nn.Sequential(
            nn.Conv3d(16, 32, kernel_size=(4, 3, 3)),
            nn.BatchNorm3d(32),
            nn.ReLU(),
            nn.MaxPool3d(kernel_size=(2, 2, 2)),
            nn.Dropout(0.5))
        self.conv3_frame = nn.Sequential(
            nn.Conv3d(32, 64, kernel_size=(3, 3, 3)),
            nn.BatchNorm3d(64),
            nn.ReLU(),
            nn.MaxPool3d(kernel_size=(2, 2, 2)),
            nn.Dropout(0.5))

        self.conv1_flow_x = nn.Sequential(
            nn.Conv3d(1, 16, kernel_size=(3, 3, 3)),
            nn.BatchNorm3d(16),
            nn.ReLU(),
            nn.MaxPool3d(kernel_size=(1, 2, 2)),
            nn.Dropout(0.5))
        self.conv2_flow_x = nn.Sequential(
            nn.Conv3d(16, 32, kernel_size=(3, 3, 3)),
            nn.BatchNorm3d(32),
            nn.ReLU(),
            nn.MaxPool3d(kernel_size=(2, 2, 2)),
            nn.Dropout(0.5))
        self.conv3_flow_x = nn.Sequential(
            nn.Conv3d(32, 64, kernel_size=(3, 3, 3)),
            nn.BatchNorm3d(64),
            nn.ReLU(),
            nn.MaxPool3d(kernel_size=(2, 2, 2)),
            nn.Dropout(0.5))

        self.conv1_flow_y = nn.Sequential(
            nn.Conv3d(1, 16, kernel_size=(3, 3, 3)),
            nn.BatchNorm3d(16),
            nn.ReLU(),
            nn.MaxPool3d(kernel_size=(1, 2, 2)),
            nn.Dropout(0.5))
        self.conv2_flow_y = nn.Sequential(
            nn.Conv3d(16, 32, kernel_size=(3, 3, 3)),
            nn.BatchNorm3d(32),
            nn.ReLU(),
            nn.MaxPool3d(kernel_size=(2, 2, 2)),
            nn.Dropout(0.5))
        self.conv3_flow_y = nn.Sequential(
            nn.Conv3d(32, 64, kernel_size=(3, 3, 3)),
            nn.BatchNorm3d(64),
            nn.ReLU(),
            nn.MaxPool3d(kernel_size=(2, 2, 2)),
            nn.Dropout(0.5))

        self.fc1 = nn.Linear(3328, 128)
        self.dropfc1 = nn.Dropout(0.5)
        self.fc2 = nn.Linear(128, 6)

    def forward(self, frames, flow_x, flow_y):
        out_frames = self.conv1_frame(frames)
        out_frames = self.conv2_frame(out_frames)
        out_frames = self.conv3_frame(out_frames)
        out_frames = out_frames.view(out_frames.size(0), -1)

        out_flow_x = self.conv1_flow_x(flow_x)
        out_flow_x = self.conv2_flow_x(out_flow_x)
        out_flow_x = self.conv3_flow_x(out_flow_x)
        out_flow_x = out_flow_x.view(out_flow_x.size(0), -1)

        out_flow_y = self.conv1_flow_y(flow_y)
        out_flow_y = self.conv2_flow_y(out_flow_y)
        out_flow_y = self.conv3_flow_y(out_flow_y)
        out_flow_y = out_flow_y.view(out_flow_y.size(0), -1)

        out = torch.cat([out_frames, out_flow_x, out_flow_y], 1)
        out = self.fc1(out)
        out = nn.ReLU()(out)
        out = self.dropfc1(out)
        out = self.fc2(out)

        return out
```

Training and Validating:

Training data is the set of the data on which the actual training takes place. Validation split helps to improve the model performance by fine-tuning the model after each epoch. The test set informs us about the final accuracy of the model after completing the training phase.
After the 3 models were built, they are trained with their respective training data which was pre-processed earlier and stored in ‘train.p’ for 2d CNN and 3d CNN and ‘train_flow.p’ for 3d CNN with the Optical flow model. Immediately after training the models, they are validated by the validation dataset which was pre-processed earlier and stored in ‘dev.p’.
In 2d CNN the model is trained by sending each frame as an input to the model. While in 3d CNN the model is trained by passing a block of frames (Ex.15 frames.) together to the model and in 3d CNN – Optical flow model, the model is first trained by passing a block of frames as input and then, the optical flow data which was also embedded in ‘train_flow.p’ dataset.
Then each model produces train accuracy, validation accuracy, train loss and validation loss for each epoch.

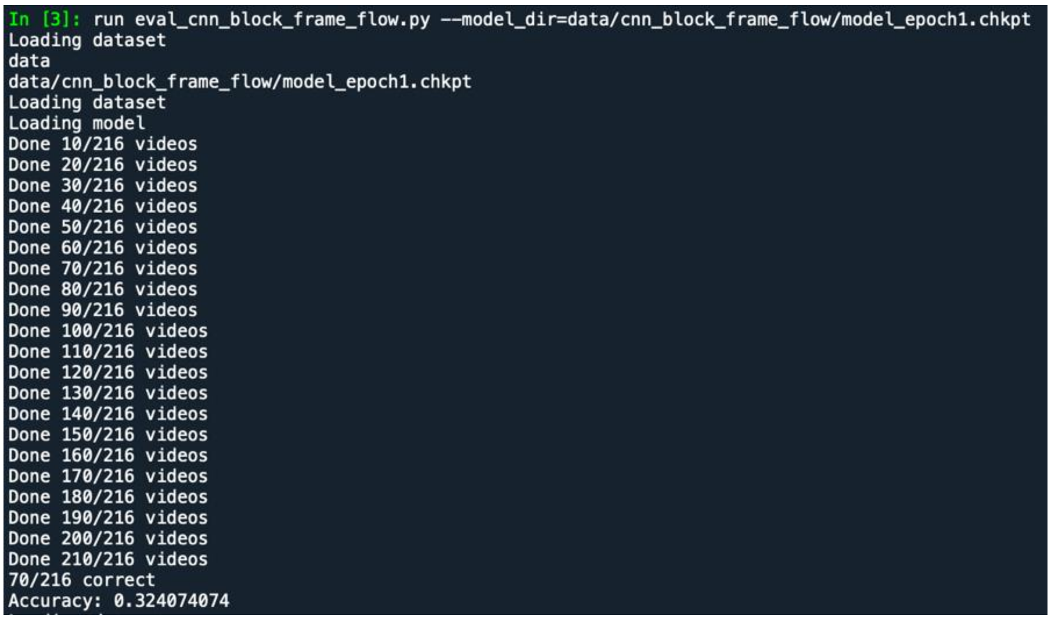


```
In [2]: run train_cnn.py --batch_size=1000 --start_epoch=1 --num_epochs=2 --cuda=0
data
Loading dataset
Start training
epoch 1/2, iteration 10/74, loss: tensor(1.7596)
epoch 1/2, iteration 20/74, loss: tensor(1.6982)
epoch 1/2, iteration 30/74, loss: tensor(1.6860)
epoch 1/2, iteration 40/74, loss: tensor(1.6105)
epoch 1/2, iteration 50/74, loss: tensor(1.6402)
epoch 1/2, iteration 60/74, loss: tensor(1.6067)
epoch 1/2, iteration 70/74, loss: tensor(1.6080)
/opt/anaconda3/envs/projects/lib/python3.8/site-packages/torch/nn/_reduction.py:42:
UserWarning: size_average and reduce args will be deprecated, please use reduction='sum'
instead.
  warnings.warn(warning.format(ret))
epoch 1/2, train_loss = tensor(1.6681), traic_acc = tensor(0.3740), dev_loss =
tensor(1.6847), dev_acc = tensor(0.3509)
[{'train_loss': tensor(1.6681), 'train_acc': tensor(0.3740), 'dev_loss': tensor(1.6847),
'dev_acc': tensor(0.3509)}]
epoch 2/2, iteration 10/74, loss: tensor(1.5478)
epoch 2/2, iteration 20/74, loss: tensor(1.4759)
epoch 2/2, iteration 30/74, loss: tensor(1.4489)
epoch 2/2, iteration 40/74, loss: tensor(1.4158)
epoch 2/2, iteration 50/74, loss: tensor(1.4133)
epoch 2/2, iteration 60/74, loss: tensor(1.3559)
epoch 2/2, iteration 70/74, loss: tensor(1.3490)
epoch 2/2, train_loss = tensor(1.2171), traic_acc = tensor(0.5178), dev_loss =
tensor(1.2554), dev_acc = tensor(0.5005)
```

Model Testing:

A model might fit the training dataset perfectly well. But there is no guarantee that it will do equally well in real life? In order to assure that, select samples for testing set from the overall dataset — examples that the machine hasn't seen before. It is important to remain unbiased during selection and draw samples at random. Also, should not use the same set many times to avoid training on your test data. The test set should be large enough to provide statistically meaningful results and be representative of the data set as a whole.

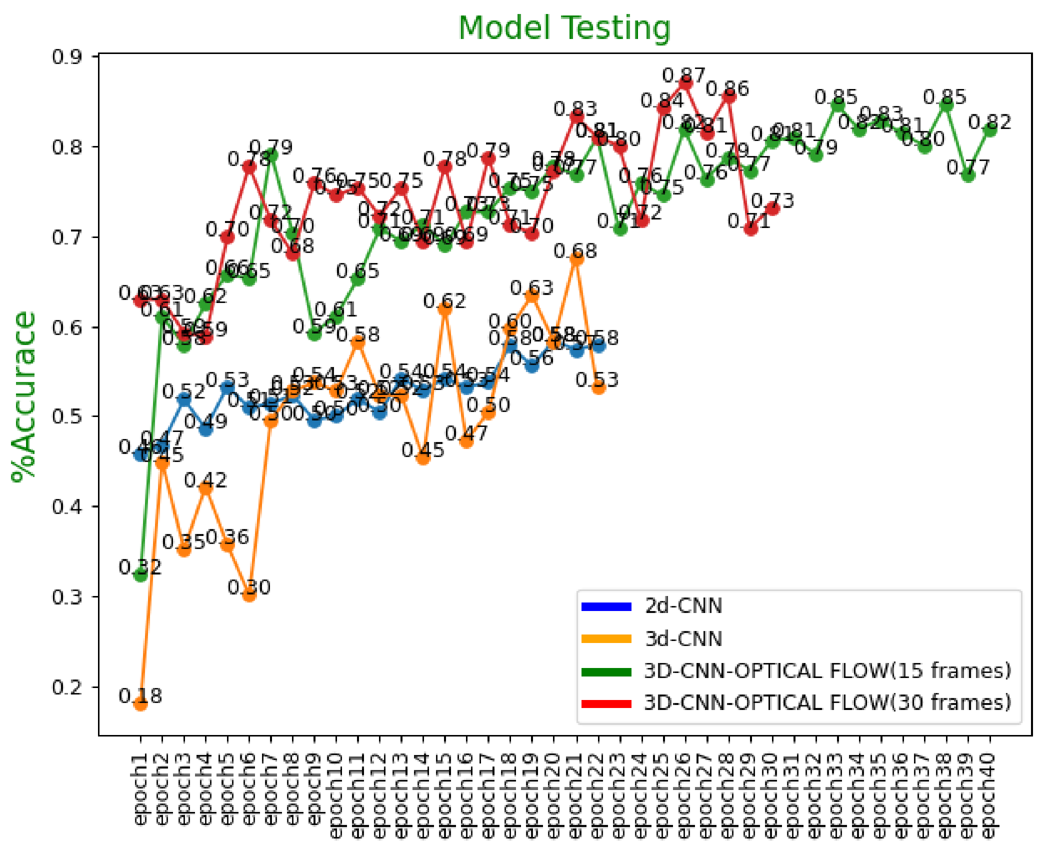
After the models were trained and validated, the models are tested by pre-processed test datasets. 2d CNN and 3d CNN models are tested with 'test.p' dataset and the 3d CNN – optical flow model is tested with 'test_flow.p' dataset.



```
In [3]: run eval_cnn_block_frame_flow.py --model_dir=data/cnn_block_frame_flow/model_epoch1.chkpt
Loading dataset
data
data/cnn_block_frame_flow/model_epoch1.chkpt
Loading dataset
Loading model
Done 10/216 videos
Done 20/216 videos
Done 30/216 videos
Done 40/216 videos
Done 50/216 videos
Done 60/216 videos
Done 70/216 videos
Done 80/216 videos
Done 90/216 videos
Done 100/216 videos
Done 110/216 videos
Done 120/216 videos
Done 130/216 videos
Done 140/216 videos
Done 150/216 videos
Done 160/216 videos
Done 170/216 videos
Done 180/216 videos
Done 190/216 videos
Done 200/216 videos
Done 210/216 videos
70/216 correct
Accuracy: 0.324074074
```

Results:

After testing the model, the test accuracies were noted at each epoch and plotted using a 'Pyplot' to determine the best model.


Model Testing
%Accurace
2d-CNN
3d-CNN
3D-CNN-OPTICAL FLOW(15 frames)
3D-CNN-OPTICAL FLOW(30 frames)
epoch1 … epoch40