Skeleton-based Action Recognition - Americal Sign Language (ASL) using fused ConvLSTM - LSTM

Eashan Kaushik,[1] **Rishab Singh** [2]

[1] New York University
[2] New York University
ek3575@nyu.edu, rs7623@nyu.edu

Abstract

Action recognition is a challenging problem in deep learning. With the advancement in Deep Learning, ConvLSTM (8) and LSTMs have been widely used for action recognition. Both these approaches show promising results; in this project, we will implement an architecture by fusing these two architectures. We will also take advantage of the human skeleton-based approach in action recognition that takes advantage of a compact representation of human action.

Introduction

American Sign Language (ASL) is widely spread within the deaf community as the primary source of communication with others, being used in 40 countries and containing more than 10,000 phrases. However, only 1% of the population knows sign language (1). Hence, recognizing American Sign Language (ASL) has various real-world applications. We have implemented a fused ConvLSTM and LSTM architecture in this project to detect 10 ASL signs. We successfully achieved a test accuracy of 0.901 on the custom ASL dataset. To compare our architecture with other Action Rekognition architecture, we have also trained our model on UCF-YouTube Action Dataset, achieving a test accuracy of 0.77. Human pose contains valuable information about ongoing human actions and can be combined with video frames or separately to detect actions. To detect and track human pose, we will make use of MediaPipe (2).

Public GitHub repository: Action Recognition Repo
Dataset: UCF-Youtube Action, ASL Dataset

Literature Survey

Recently there has been a lot of development in the field of action recognition. This section will discuss various approaches to recognition actions in videos. We essentially can take advantage of pre-trained architectures like VGG16, InceptionV3, ResNet50, Xception, and MobileNet V2 models for action recognition (6), by converting video data into image data. However, this approach results in the loss of available video frames, which might be essential in accurately classifying the action. To process video data 3-D, Convolutional Neural Networks (3D-CNN) have been widely used. This involves depth-wise convolutions rather than 2-D convolutions, allowing us to process video data. One such architecture, C3D (7), contains 8 convolutional, five pooling layers, and two fully connected layers. 3-D CNN gives us richer motion information, however, fails to capture the temporal relationship between frames. ConvLSTM (15) can also capture temporal relationships with Video-Based Recurrent Neural Networks. This model achieves 75.26% (Cross Subject- CS) and 75.45% (Cross View -CV) for the stateless model and 80.43% (CS) and 79.91%(CV) for the stateful version on the NTU RGB+D dataset.

All the architectures discussed till now fail to accommodate human skeleton information while recognizing actions. Human skeletons or key points contain valuable information about human intent. Key points can be divided into face, left-hand, right-hand, and pose landmarks. These key points can either be used as heatmaps, imposed on RGB frames, or as vectors for training and processing. MediaPipe (2) offers cross-platform, customizable ML solutions for live and streaming media and is widely used to generate human key points. One approach involves stacking 2D joint heatmaps, 2D limb heatmaps, and RGB images for feeding through a 3D CNN recognized as PoseConv3D (3). PoseConv3D achieves an accuracy of 84.8% accuracy on the NTU-120 dataset with 2.8 million parameters.

However, as discussed earlier, we can also take advantage of key points in vector forms. All the key points of a single frame can be stacked together and fed into an LSTM network for all timestamps; this is explored in Spatio-Temporal LSTM Network with Trust Gates - ST-LSTM (10). This paper implements fusion between LSTM cells and results in 69.2% accuracy on the NTU RGB+D dataset. However, it can be argued that fusion between CNN and LSTM works better can fusion between LSTM and LSTM, even though CNN essentially leads to loss of temporal information. This is explored in SPF-LSTM (4). This approach achieved an accuracy of 87.40% on NTU RGB+D, a major improvement over non-skeleton approaches discussed earlier.

SpatialTemporal Graph Convolutional Networks (ST-GCN) (9) is another architecture that has shown promising results achieving a competitive 88.3% accuracy on the NTU RGB+D dataset. ST-GCN uses graph convolution to predict human actions from a skeleton sequence by computing a spatial, temporal graph. This provides an advantage of faster computation than conventional 3D convolution for action detection because it can only estimate actions based on skeletal information.

In this project, we explore fusing ConvLSTM with LSTM networks for action recognition. We essentially want to develop an architecture that can provide action predictions in real-time and, at the same time, is quite accurate for ASL recognition. We will first explore the dataset, followed by the architecture and results.

Dataset

Initially, we will test our model on UCF-YouTube Action Data Set; it contains 11 action categories: basketball shooting, biking/cycling, diving, golf swinging, horseback riding, soccer juggling, swinging, tennis swinging, trampoline jumping, volleyball spiking, and walking with a dog. Finally, we have developed our dataset for ASL, consisting of 10 classes: yes, thanks, please, no, mother, me, I love you, help, hello, and father.

We will use left-hand, right-hand, face, and pose landmarks from MediaPipe. For each frame, we get 468 face landmarks, 33 pose landmarks, 21 left-hand, and 21 right-hand landmarks. Face left-hand and right-hand landmarks have three coordinates (x, y, z); pose landmarks, on the other, are represented by four coordinates (x, y, z, visibility). A total of (468 3) + (21 3) 2 + (33 4) = 1662 key points per frame are utilized.

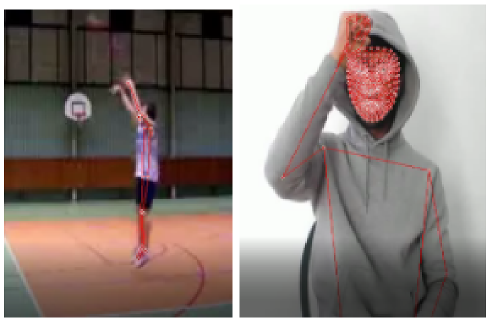
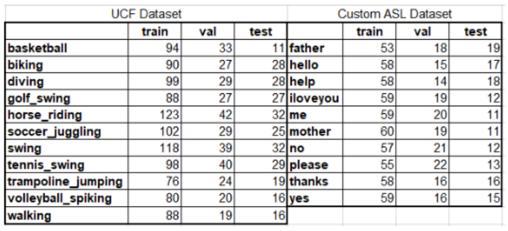


Figure 1: Sample instance of (right) UCF dataset and (left) custom dataset

For UCF-YouTube Action Data Set, we will use a sequence length of 20, i.e., each video will be of a shape (3, 20, 64, 64), where 3 represents the RGB channels, 20 is the sequence length, and 64 height and width respectively. Consequently, we will have a feature vector of shape (20, 1662) representing the key points.

For our custom dataset, we will use a sequence length of 30 frames, i.e., each video will be of a shape (3, 30, 64, 64), where 3 represents the RGB channels, 30 is the sequence length, and 64 height and width, respectively. Consequently, we will have a feature vector of shape (30, 1662) representing the key points. We have used random sampling to take out the desired number of frames from train, test, and validation data. For the UCF dataset, we have 1056 training instances, 329 validation instances, and 263 testing instances, and for the custom dataset, we have 576 training instances, 180 validation instances, and 144 testing instances. Class-wise splits for both these datasets can be seen below:



UCF Dataset				Custom ASL Dataset			
	train	val	test		train	val	test
basketball	94	33	11	father	53	18	19
biking	90	27	28	hello	58	15	17
diving	99	29	28	help	58	14	18
golf_swing	88	27	27	iloveyou	59	19	12
horse_riding	123	42	32	me	59	20	11
soccer_juggling	102	29	25	mother	60	19	11
swing	118	39	32	no	57	21	12
tennis_swing	98	40	29	please	55	22	13
trampoline_jumping	76	24	19	thanks	58	16	16
volleyball_spiking	80	20	16	yes	59	16	15
walking	88	19	16				

Figure 2: Dataset Split

Architecture

LSTM

RNNs are a type of neural network that is able to process sequential data, such as time series or natural language. They do this by using looping connections, called recurrent connections, that allow information to be passed from one-time step to the next. Long Short-Term Memory (LSTM) is a type of Recurrent Neural Network (RNN) that is able to capture long-term dependencies in data. LSTMs can remember information for longer periods using a set of special units called ”cells” that can store and manipulate information. These cells are controlled by a group of gates that determine what information is stored, what is forgotten, and what is output.

In our architecture, we have used two-layer LSTM which is ”rolled over time”. The following model shows the processing for t timestamps. In our case, t is equal to the number of frames in the video we are processing.

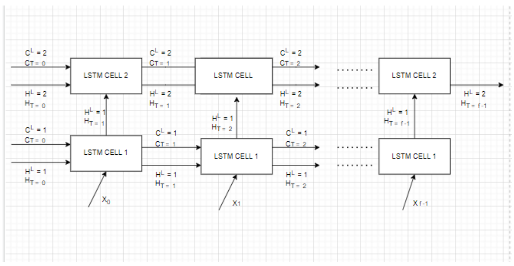


Figure 3: LSTM Rolled-Over Time

ConvLSTM

Convolutional Long Short-Term Memory (ConvLSTM) is a type of neural network architecture that combines the capabilities of Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTMs). It is a variant of LSTM that is able to process both spatial and temporal data and is particularly well-suited for tasks such as video prediction and anomaly detection.

ConvLSTMs use the same concept of cells and gates as regular LSTMs but also include convolutional layers that allow them to process spatial information. The convolutional

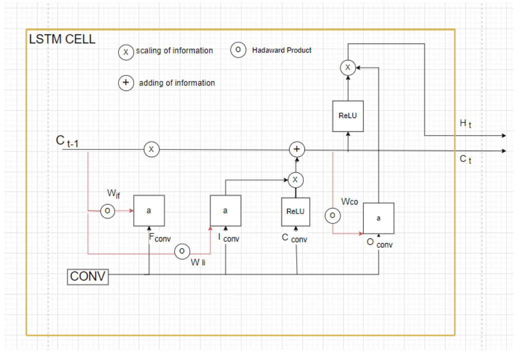


Figure 4: ConvLSTM Cell

layers operate on the input data in a similar way to how they do in a CNN, by learning local patterns and features. The LSTM part of the network can then process the temporal information and use the learned features to make predictions about future frames in a video or identify anomalies in a time series.

In **Fig. 4**, we have shown our implementation of convLSTM borrowed from (16). We have implemented this using the following equations. In this equation, X_t is the t^{th} frame, C_t is the cell state, and h_t represents the hidden state. F, i, and O represent the forget gate, input gate, and output gate.

$$\mathrm{X}_t^{\times} = concatenate([X_t, h_t - 1])$$

$$\mathrm{X}_{conv}^{\times} = X_t^{\times} w + b$$

$$\mathrm{F}_{conv}, i_{conv}, C_{conv}, O_{conv} = chunks(X_{conv}^{\times})$$

$$\mathrm{F}_t = a(C_{t-1} W_{cf} + F_{conv})$$

$$\mathrm{i}_t = a(C_{t-1} W_{ci} + i_{conv})$$

$$\mathrm{C}_t = F_t * C_{t-1} + i_t * ReLU(C_{conv}))$$

$$\mathrm{O}_t = a(C_t W_c 0 + O_{conv})$$

$$\mathrm{h}_t = O_t * ReLU(C_t)$$

Fused ConvLSTM and LSTM

This architecture fuses the output of both the ConvLSTM model and the LSTM model for training and evaluation. **Fig. 5** shows us the fused architecture.

Training

- Hyperparameters: This model consists of 7174718 trainable parameters. While training, we used a batch size of 64, a learning rate of 0.0001, and a weight decay (L-2 regularization) of 0.8. We set these hyperparameters after various experiments as these showed the best results in a shorter number of epochs. We have a high value of weight decay as our model was overfitting roughly after 50 epochs. Higher weight decay ensures we do not overfit the data and generalize well to unseen data.

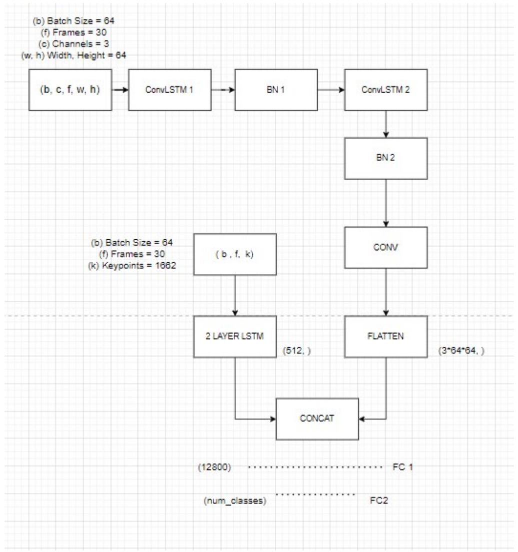


Figure 5: Fused ConvLSTM and LSTM

- Optimizer and Learning rate scheduler: We experimented with two optimizers, i.e., Adam optimizer and Stochastic Gradient Descent optimizer with a momentum of 0.9. We ended up using adam optimizer as it could generalize well to unseen data, i.e., we witnessed slightly lower validation loss. We also used a multi-step learning rate scheduler to decay our learning rate by 0.1 at multiple steps during training. Stacking Batch Normalization and Activation layers properly is very important.
- Loss function : Since the task of action recognition falls in the classification domain, we used the cross-entropy loss function. It is a measure of the difference between the predicted probability distribution and the true class distribution and is calculated as the negative log-likelihood of the true class given the predicted probability distribution. Cross entropy loss tries to minimize the distance between two probabilities if it is a correct classification; otherwise, it minimizes the distance.

$$L = \frac{1}{N} \sum_{x} y.log(y))$$

where L is the cross-entropy loss, y' is the true label, and y is the predicted probability for the true label.
- Batch Normalization: Batch normalization solves a major problem called internal covariate shift, and ReLu Activation functions can help with the vanishing gradient problem. We have used the Batch Normalization layer after each ConvLSTM cell to normalize the contributions to a layer for every mini-batch.
- Weight Intialization: We take advantage of weight initialization to prevent exploding and vanishing gradient problems during training. We initialized convolution layers and fully connected layers with Xavier's normal initialization and filled the bias layer with 0.01. Hidden states and cell states are initialized with zeros.

Results

We trained the UCF-YouTube Action dataset for 200 epochs. The loss and accuracy training curves and metrics can be seen below:

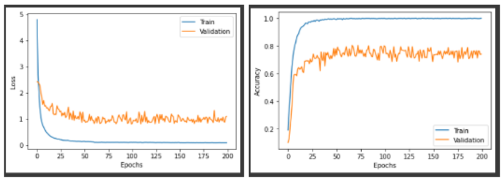


Figure 6: Loss-training curve and Accuracy-training curve

	Loss	Accuracy
Train	0.09625	0.999081
Validation	0.86132	0.7875
Test	-	**0.777679**

Table 1: Loss and Accuracy - UCF-YouTube Action dataset

We have also summarized the results using the confusion matrix on the test dataset. We can see that most of the classes were correctly recognized, except for walking and biking. Further training and more data, can be used to increase the performance of the test dataset.

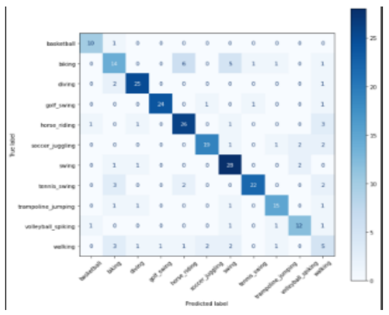


Figure 7: Confusion Matrix - UCF-YouTube Action dataset

We trained for the Custom ASL dataset 200 epochs. The loss and accuracy training curves and metrics can be seen **Fig. 8**. The confusion matrix on the test dataset on this dataset can be visualized **Ref. 9**. We can see that most of the classes were correctly recognized.

Conclusion

In this section, we will discuss what we initially intended to complete and what we accomplished. We intended to achieve 70% test accuracy on the UCF-YouTube Action dataset and accomplished an accuracy of 77%. We intended to achieve 90% test accuracy on the ASL dataset that we have created and accomplished this task, we combined training and evaluation of LSTM and ConvLSTM models to take advantage of both these architectures as proposed and provided reasonable explanations for combing these two architectures.

	Loss	Accuracy
Train	0.104224	0.994792
Validation	0.885417	0.885417
Test	-	**0.901042**

Table 2: Loss and Accuracy Custom American Sign Language dataset

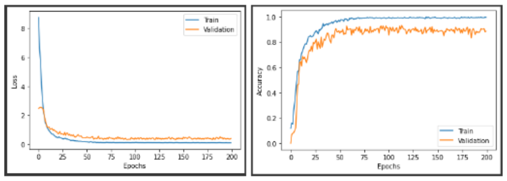


Figure 8: Loss-training curve and Accuracy-training curve

References

[1] Lacke, Susan. "Do All Deaf People Use Sign Language?" Do All Deaf People Use Sign Language?, 5 Aug. 2020, www.accessibility.com/blog/do-all-deaf-people-use-sign-language.

[2] LLC, Google. "Home." Mediapipe, google.github.io/mediapipe. Accessed 17 Dec. 2022.

[3] Duan, Haodong, et al. "Revisiting skeleton-based action recognition." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2022.

[4] Li, Chuankun, et al. "Skeleton-based action recognition using LSTM and CNN." 2017 IEEE International Conference on Multimedia & Expo Workshops (ICMEW). IEEE, 2017.

[5] Bhosale, Radhika. "Skeleton-based American Sign Language Recognition." ScholarWorks, 5 Aug. 2020, scholarworks.calstate.edu/concern/theses/4m90f113t?locale=en.

[6] Apon, Tasnim Sakib, et al. "Real Time Action Recognition from Video Footage." 2021 3rd International Conference on Sustainable Technologies for Industry 4.0 (STI). IEEE, 2021.

[7] D. Tran, L. Bourdev, R. Fergus, L. Torresani and M. Paluri, "Learning Spatiotemporal Features with 3D Convolutional Networks," 2015 IEEE International Conference on Computer Vision (ICCV), 2015, pp. 4489-4497, doi: 10.1109/ICCV.2015.510.

[8] Xingjian Shi, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-kin Wong, and Wang-chun Woo. 2015. Convolutional LSTM Network: a machine learning approach for precipitation nowcasting. In Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'15). MIT Press, Cambridge, MA, USA, 802–810.

[9] Yan, S., Y. Xiong, and D. Lin. "Spatial Temporal Graph Convolutional Networks for Skeleton-Based Action Recognition". Proceedings of the AAAI Conference

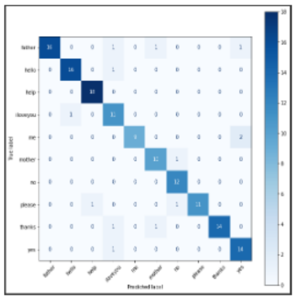


Figure 9: Confusion Matrix -Custom American Sign Language dataset

on Artificial Intelligence, vol. 32, no. 1, Apr. 2018, doi:10.1609/aaai.v32i1.12328.

[10] Liu J, Shahroudy A, Xu D, Kot AC, Wang G. Skeleton-Based Action Recognition Using Spatio-Temporal LSTM Network with Trust Gates. IEEE Trans Pattern Anal Mach Intell. 2018 Dec;40(12):3007-3021. doi: 10.1109/TPAMI.2017.2771306. Epub 2017 Nov 9. PMID: 29990167.

[11] ndplz. "GitHub - Ndrplz/ConvLSTM_Pytorch: Implementation of Convolutional LSTM in PyTorch." GitHub, 24 Mar. 2020, github.com/ndrplz/ConvLSTM_pytorch.

[12] Panda, Rohit. "Video Frame Prediction Using ConvLSTM Network in PyTorch." Medium, 11 July 2021, sladewinter.medium.com/video-frame-prediction-using-convlstm-network-in-pytorch-b5210a6ce582.

[13] Anwar, Taha, et al. "Human Activity Recognition Using TensorFlow (CNN + LSTM) — Bleed AI." Bleed AI, 24 Sept. 2021, bleedai.com/human-activity-recognition-using-tensorflow-cnn-lstm.

[14] AI, Bleed. "Human Activity Recognition Using TensorFlow (CNN + LSTM) — 2 Methods." YouTube, 24 Sept. 2021, www.youtube.com/watch?v=QmtSkq3DYko.
lm15 Renotte, Nicholas. "Sign Language Detection Using ACTION RECOGNITION With Python — LSTM Deep Learning Model." YouTube, 18 June 2021, www.youtube.com/watch?v=doDUihpj6ro.
lm16 López, Fernando. "From a LSTM Cell to a Multilayer LSTM Network With PyTorch." Medium, 28 July 2020, towardsdatascience.com/from-a-lstm-cell-to-a-multilayer-lstm-network-with-pytorch-2899eb5696f3.

[15] Sanchez-Caballero, Adrian, David Fuentes-Jimenez, and Cristina Losada-Gutiérrez. "Exploiting the convlstm: Human action recognition using raw depth video-based recurrent neural networks." arXiv preprint arXiv:2006.07744 (2020).

[16] Ndrplz. (n.d.). Ndrplz/convlstm$_{pytorch}$: $ImplementationofConvolutionalLSTMinpytorch.GitHub.RetrievedDecember$18, 2022, $fromhttps: //github.com/ndrplz/ConvLSTM\_pytorch$

[17] Panda, R. (2021, July 11). Video frame prediction using CONVLSTM network in pytorch. Medium. Retrieved December 18, 2022, from https://sladewinter.medium.com/video-frame-prediction-using-convlstm-network-in-pytorch-b5210a6ce582

Check Runtime

```
gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
if gpu_info.find('failed') >= 0:
  print('Not connected to a GPU')
else:
  print(gpu_info)
```

```
Thu Dec 15 03:19:45 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   50C    P0    26W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

```
from psutil import virtual_memory
ram_gb = virtual_memory().total / 1e9
print('Your runtime has {:.1f} gigabytes of available RAM\n'.format(ram_gb))

if ram_gb < 20:
  print('Not using a high-RAM runtime')
else:
  print('You are using a high-RAM runtime!')
```

```
Your runtime has 27.3 gigabytes   f available RAM

You are using a high-RAM runtime!
```

Import Libraries

[ ] ↳ *2 cells hidden*

Global Variables

```
SEED = 42

np.random.seed(SEED)
random.seed(SEED)

BATCH_SIZE=64

LEARNING_RATE=0.0001
WEIGHT_DECAY=0.8
# SGD
MOMENTUM=0.9

IMAGE_HEIGHT , IMAGE_WIDTH = 64, 64

FRAMES_PER_VIDEO = 30

ROOT_DIR=os.path.join('drive', 'MyDrive', 'Custom-Dataset')
DEVICE = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
# DEVICE = "cpu"
```

Dataset

```
from google.colab import drive
drive.mount('/content/drive')
```

```
    Mounted at /content/drive
```

```
X_train_frames = np.load(os.path.join(ROOT_DIR, "train", "X_train_frames.npy"))
X_train_keypoints = np.load(os.path.join(ROOT_DIR, "train", "X_train_keypoints.npy"))
y_train = np.load(os.path.join(ROOT_DIR, "train", "y_train.npy"))

X_test_frames = np.load(os.path.join(ROOT_DIR, "test", "X_test_frames.npy"))
X_test_keypoints = np.load(os.path.join(ROOT_DIR, "test", "X_test_keypoints.npy"))
y_test = np.load(os.path.join(ROOT_DIR, "test", "y_test.npy"))

X_val_frames = np.load(os.path.join(ROOT_DIR, "val", "X_val_frames.npy"))
X_val_keypoints = np.load(os.path.join(ROOT_DIR, "val", "X_val_keypoints.npy"))
y_val = np.load(os.path.join(ROOT_DIR, "val", "y_val.npy"))
```

```
with open(os.path.join(ROOT_DIR, "label_encoder.json")) as f:
    data = json.load(f)
CLASSES = list(data.keys())
```

```
# custom dataset class
class UCFCustom(Dataset):
    def __init__(self, frames, keypoints, labels, transform=None):
        self.frames = frames
        self.labels = labels
        self.keypoints = keypoints
        self.transform = transform

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, index):
        X_frames = np.moveaxis(self.frames[index], 3, 0) # moving axis for proper orientation (channels, frames, height, width)
        X_frames = torch.from_numpy(X_frames) # convert X_frames to tensor
        X_keypoints = torch.from_numpy(self.keypoints[index]) # convert X_keypoints to tensor
        y = torch.from_numpy(np.array(self.labels[index])).type(torch.LongTensor) # convert y to tensor

        X_frames = X_frames.float()
        X_keypoints = X_keypoints.float()
        # y = y.float()


        if self.transform: # if transform
            X_frames = self.transform(X_frames)
            X_keypoints = self.transform(X_keypoints)

        return X_frames, X_keypoints, y # return frame, keypoint and y
```

```
test_dataset = UCFCustom(X_test_frames, X_test_keypoints, y_test, transform=None)
val_dataset = UCFCustom(X_val_frames, X_val_keypoints, y_val, transform=None)
train_dataset = UCFCustom(X_train_frames, X_train_keypoints, y_train, transform=None)
```

```
test_dataloader = DataLoader(test_dataset, batch_size=BATCH_SIZE,
                                          shuffle=True, num_workers=0)
train_dataloader = DataLoader(train_dataset, batch_size=BATCH_SIZE,
                                          shuffle=True, num_workers=0)
val_dataloader = DataLoader(val_dataset, batch_size=BATCH_SIZE,
                                          shuffle=True, num_workers=0)
```

```
X_frames, X_keypoints, y = next(iter(train_dataloader))
```

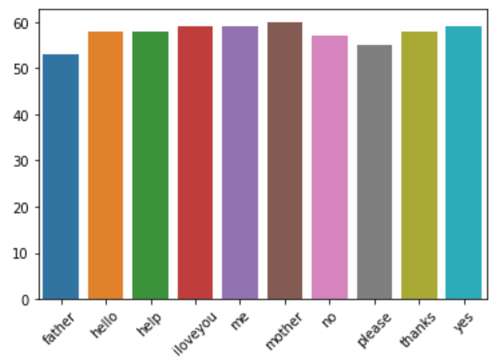
```
X_frames.shape
```

```
    torch.Size([64, 3, 30, 64, 64])
```

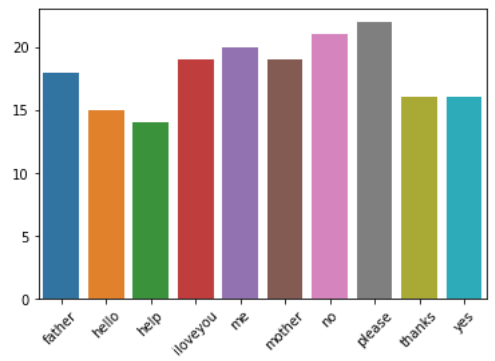
```
X_keypoints.shape
```

```
    torch.Size([64, 30, 1662])
```

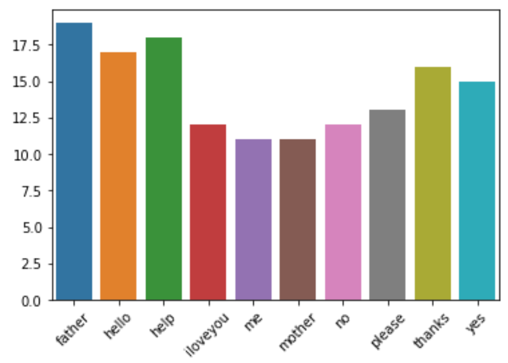
```
sample = dict()
x_axis, y_axis = np.unique(y_train, return_counts=True)
sns.barplot(x=CLASSES, y=y_axis)
plt.xticks(rotation=45)
plt.show()
sample["train"] = list(y_axis)
```



```
x_axis, y_axis = np.unique(y_val, return_counts=True)
sns.barplot(x=CLASSES, y=y_axis)
plt.xticks(rotation=45)
plt.show()
sample["val"] = list(y_axis)
```



```
x_axis, y_axis = np.unique(y_test, return_counts=True)
sns.barplot(x=CLASSES, y=y_axis)
plt.xticks(rotation=45)
plt.show()
sample["test"] = list(y_axis)
```



```
import pandas as pd

df = pd.DataFrame(data=sample, index=CLASSES)

from google.colab import files
df.to_csv('filename.csv')
files.download('filename.csv')
```

Model

Module

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

    def loss(self, y_hat, y):
        loss = nn.CrossEntropyLoss()
        return loss(y_hat, y)

    def optimizer(self, optimizer_type="SGD"):
        assert optimizer_type in ("SGD", "Adam"), f"optimizer_type must be SGD or Adam not {optimizer_type}"
        assert hasattr(self, "lr"), "Learning rate is not defined"
        assert hasattr(self, "weight_decay"), "Weight Decay rate is not defined"

        # SGD or Adam optimizer
        if optimizer_type == "SGD":
            assert hasattr(self, "momentum"), "Momentum is not defined"
            optimizer = torch.optim.SGD(self.parameters(), lr=self.lr, momentum=self.momentum, weight_decay=self.weight_decay)
        else:
            optimizer = torch.optim.Adam(self.parameters(), lr=self.lr, weight_decay=self.weight_decay)

        # step scheduler
        scheduler = lr_scheduler.MultiStepLR(optimizer, milestones=[60, 150], gamma=0.1)

        return optimizer, scheduler

    def step(self, X_frames, X_keypoints, y):
        y_pred = self(X_frames, X_keypoints) # prediction
        return self.loss(y_pred, y), y_pred # loss
```

ConvLSTM

```
# Original ConvLSTM cell as proposed by Shi et al.
class ConvLSTMCell(nn.Module):

    def __init__(self, in_channels, out_channels,
    kernel_size, padding, activation, frame_size):

        super(ConvLSTMCell, self).__init__()

        if activation == "tanh":
            self.activation = torch.tanh
        elif activation == "relu":
            self.activation = torch.relu

        # Idea adapted from https://github.com/ndrplz/ConvLSTM_pytorch
        self.conv = nn.Conv2d(
            in_channels=in_channels + out_channels,
            out_channels=4 * out_channels,
            kernel_size=kernel_size,
            padding=padding)

        #·xavier·initializtion
        torch.nn.init.xavier_normal_(self.conv.weight)
        self.conv.bias.data.fill_(0.01)

        # Initialize weights for Hadamard Products
        self.W_ci = nn.Parameter(torch.Tensor(out_channels, *frame_size))
        self.W_co = nn.Parameter(torch.Tensor(out_channels, *frame_size))
        self.W_cf = nn.Parameter(torch.Tensor(out_channels, *frame_size))

    def forward(self, X, H_prev, C_prev):

        # Idea adapted from https://github.com/ndrplz/ConvLSTM_pytorch
        conv_output = self.conv(torch.cat([X, H_prev], dim=1))

        # Idea adapted from https://github.com/ndrplz/ConvLSTM_pytorch
        i_conv, f_conv, C_conv, o_conv = torch.chunk(conv_output, chunks=4, dim=1)

        input_gate = torch.sigmoid(i_conv + self.W_ci * C_prev )
        forget_gate = torch.sigmoid(f_conv + self.W_cf * C_prev )

        # Current Cell output
        C = forget_gate*C_prev + input_gate * self.activation(C_conv)

        output_gate = torch.sigmoid(o_conv + self.W_co * C )

        # Current Hidden State
        H = output_gate * self.activation(C)

        return H, C
```

```
class ConvLSTM(nn.Module):

    def __init__(self, in_channels, out_channels,
    kernel_size, padding, activation, frame_size):

        super(ConvLSTM, self).__init__()

        self.out_channels = out_channels

        # We will unroll this over time steps
        self.convLSTMcell = ConvLSTMCell(in_channels, out_channels,
        kernel_size, padding, activation, frame_size)

    def forward(self, X):

        # X is a frame sequence (batch_size, num_channels, seq_len, height, width)

        # Get the dimensions
        batch_size, _, seq_len, height, width = X.size()

        # Initialize output
        output = torch.zeros(batch_size, self.out_channels, seq_len,
        height, width, device=DEVICE)

        # Initialize Hidden State
        H = torch.zeros(batch_size, self.out_channels,
        height, width, device=DEVICE)

        # Initialize Cell Input
        C = torch.zeros(batch_size,self.out_channels,
        height, width, device=DEVICE)

        # Unroll over time steps
        for time_step in range(seq_len):

            H, C = self.convLSTMcell(X[:,:,time_step], H, C)

            output[:,:,time_step] = H

        return output
```

Architecture

```
class Seq2Seq(Module):

    def __init__(self, num_channels, num_kernels, kernel_size, padding,
    activation, frame_size, num_layers, sequence_length, keypoints_length, lr=LEARNING_RATE, momentum=MOMENTUM, weight_decay=WEIGHT_DECAY

        super(Seq2Seq, self).__init__()

        # initializing hyper parameters
        self.lr = lr
        self.momentum = momentum
        self.weight_decay = weight_decay
        self.keypoints_length = keypoints_length
        self.sequence_length = sequence_length

        self.sequential = nn.Sequential()

        ### Conv-LSTM ###
        # Add First layer (Different in_channels than the rest) - convLSTM
        self.sequential.add_module(
            "convlstm1", ConvLSTM(
                in_channels=num_channels, out_channels=num_kernels,
                kernel_size=kernel_size, padding=padding,
                activation=activation, frame_size=frame_size)
        )


        self.sequential.add_module(
            "batchnorm1", nn.BatchNorm3d(num_features=num_kernels)
        )

        # Add rest of the layers - convLSTM
        for l in range(2, num_layers+1):

            self.sequential.add_module(
                f"convlstm{l}", ConvLSTM(
                    in_channels=num_kernels, out_channels=num_kernels,
                    kernel_size=kernel_size, padding=padding,
                    activation=activation, frame_size=frame_size)
                )


            self.sequential.add_module(
                f"batchnorm{l}", nn.BatchNorm3d(num_features=num_kernels)
                )

        self.conv = nn.Conv2d(
            in_channels=num_kernels, out_channels=num_channels,
            kernel_size=kernel_size, padding=padding)

        # xavier initializtion
        torch.nn.init.xavier_normal_(self.conv.weight)
        self.conv.bias.data.fill_(0.01)

        ### LSTM ###
        self.lstm_cell_layer_1 = nn.LSTMCell(self.keypoints_length, 512)

        self.lstm_cell_layer_2 = nn.LSTMCell(512, 512)

        self.fc = nn.Linear( (3 * 64 * 64) + 512, 11)

        # xavier initializtion
        torch.nn.init.xavier_normal_(self.fc.weight)
        self.fc.bias.data.fill_(0.01)

        self.device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

    def forward(self, X_frames, X_keypoints):

       ### Conv-LSTM ###
        output = self.sequential(X_frames)

        output = self.conv(output[:,:,-1])

        output = output.view(output.size(0), -1)

         ### LSTM ###
        hidden_state = torch.zeros(X_keypoints.size(0), 512).to(self.device)
        cell_state = torch.zeros(X_keypoints.size(0), 512).to(self.device)
        hidden_state_2 = torch.zeros(X_keypoints.size(0), 512).to(self.device)
        cell_state_2 = torch.zeros(X_keypoints.size(0), 512).to(self.device)

        # xavier initializtion 2 layer LSTM
        torch.nn.init.xavier_normal_(hidden_state)
        torch.nn.init.xavier_normal_(cell_state)
        torch.nn.init.xavier_normal_(hidden_state_2)
        torch.nn.init.xavier_normal_(cell_state_2)

        out = X_keypoints.view(self.sequence_length, X_keypoints.size(0), -1)

        for i in range(self.sequence_length):
          hidden_state, cell_state = self.lstm_cell_layer_1(out[i], (hidden_state, cell_state))
          hidden_state_2, cell_state_2 = self.lstm_cell_layer_2(hidden_state, (hidden_state_2, cell_state_2))

        final_output = torch.cat((output, hidden_state_2), 1)

        final_output = self.fc(final_output)

        return final_output
```

TRAINER

```
class Trainer():
    def __init__(self, epochs=10):
        self.epochs = epochs

    def prepare_model(self, model):
        model.trainer = self
        self.model = model
    def save_checkpoint(self, epoch):
        torch.save(self.model, os.path.join("drive/MyDrive/checkpoint", f"model_{epoch + 1}.pt"))
        with open(os.path.join("drive/MyDrive/checkpoint", f"history_{epoch + 1}.json"), 'w') as fp:
            json.dump(self.history, fp)

    def load_checkpoint(self, epoch):
        self.model = torch.load(os.path.join("drive/MyDrive/checkpoint", f"model_{epoch + 1}.pt"))
        with open(os.path.join("drive/MyDrive/checkpoint", f"history_{epoch + 1}.json"), "r") as json_file:
            self.history = json.load(json_file)

    def epoch_time(self, start_time, end_time):
        elapsed_time = start_time - end_time
        elapsed_mins = int(elapsed_time/60)
        elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
        return elapsed_mins, elapsed_secs
    def fit(self, train_dataloader, val_dataloader, model):

        self.device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
        # self.device = 'cpu'
        highest_val = 0
        val_threshold = 0.81
        self.prepare_model(model)
        self.model.to(self.device)

        self.optimizer, self.scheduler = self.model.optimizer("Adam")

        self.history = {"train": {"loss": [], "accuracy": []}, "val": {"loss": [], "accuracy": []}}
        for epoch in range(0, self.epochs):

            start_time = time.time()
            train_loss, train_acc = self.fit_train(train_dataloader)
            val_loss, val_acc = self.fit_val(val_dataloader)
            end_time = time.time()

            self.scheduler.step()

            elapsed_mins, elapsed_secs = self.epoch_time( start_time, end_time)

            print(f"<---- Epoch {epoch + 1}: {elapsed_mins}m {elapsed_secs}s ---->")
            print(f"Train Loss: {train_loss}, Train Accuracy: {train_acc}")
            print(f"Val Loss: {val_loss}, Val Accuracy: {val_acc}")

            self.history["train"]["loss"].append(train_loss)
            self.history["train"]["accuracy"].append(train_acc)
            self.history["val"]["loss"].append(val_loss)
            self.history["val"]["accuracy"].append(val_acc)

            if val_acc > val_threshold:
              if val_acc > highest_val:
                highest_val = val_acc
                self.save_checkpoint(epoch + 1)

        self.model.to("cpu")

        return self.history
    def fit_train(self, train_dataloader):
        self.model.train()

        total_loss = 0
        accuracy = 0

        batchs = 0

        for X_frames, X_keypoints, y in train_dataloader:
            batchs += 1
            X_frames, X_keypoints, y = X_frames.to(self.device), X_keypoints.to(self.device), y.to(self.device)

            loss, y_pred = self.model.step(X_frames, X_keypoints, y)

            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()

            total_loss += loss.item()
            accuracy += accuracy_score(np.argmax(y_pred.to("cpu").detach().numpy(), axis=1).reshape(-1,1), y.to("cpu").detach().numpy().r

            del X_frames, X_keypoints, y, loss

        return total_loss / batchs, accuracy / batchs
    def fit_val(self, val_dataloader):
        assert hasattr(self, "model"), "Model is not yet defined"

        self.model.eval()
        total_loss = 0
        accuracy = 0

        batchs = 0

        for X_frames, X_keypoints, y in val_dataloader:
            batchs += 1
            X_frames, X_keypoints, y = X_frames.to(self.device), X_keypoints.to(self.device), y.to(self.device)

            with torch.no_grad():
                loss, y_pred = self.model.step(X_frames, X_keypoints, y)
                total_loss += loss.item()
                accuracy += accuracy_score(np.argmax(y_pred.to("cpu").detach().numpy(), axis=1).reshape(-1,1), y.to("cpu").detach().numpy
                del X_frames, X_keypoints, y, loss

        return total_loss / batchs, accuracy / batchs

    def evaluate(self, test_dataloader):

        assert hasattr(self, "model"), "Model is not yet defined"
        if not hasattr(self, "device"):
            self.device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
        self.model.to(self.device)
        self.model.eval()
        correct = 0
        batchs = 0
        predictions = list()
        labels = list()
        for X_frames, X_keypoints, y in test_dataloader:
            batchs += 1
            X_frames, X_keypoints, y = X_frames.to(self.device), X_keypoints.to(self.device), y.to(self.device)

            with torch.no_grad():
                _, y_pred = self.model.step(X_frames, X_keypoints, y)
                y_pred = np.argmax(y_pred.to("cpu").detach().numpy(), axis=1).reshape(-1,1)
                y = y.to("cpu").detach().numpy().reshape(-1,1)
                predictions.extend(y_pred)
                labels.extend(y)
                accuracy = accuracy_score(y_pred, y)

                correct += accuracy

                del X_frames, X_keypoints, y


        return correct / batchs, np.array(predictions, dtype=object), np.array(labels, dtype=object)
```

TRAINING

```
# The input video frames are grayscale, thus single channel
model = Seq2Seq(num_channels=3, num_kernels=16,
kernel_size=(5, 5), padding=(2, 2), activation="relu",
frame_size=(64, 64), num_layers=2, sequence_length=30, keypoints_length=1662).to(DEVICE)

def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

count_parameters(model)
```

```
7174
```

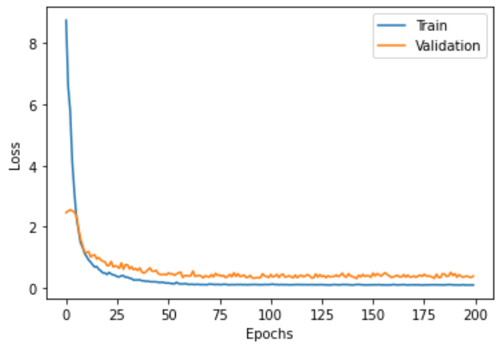
```
print(mode
```

```
eq2Seq(
 (sequential): Sequential(
   (convlstm1): ConvLSTM(
     (convLSTMcell): ConvLSTMCell(
       (conv): Conv2d(19, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
     )
   )
   (batchnorm1): BatchNorm3d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
   (convlstm2): ConvLSTM(
     (convLSTMcell): ConvLSTMCell(
       (conv): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
     )
   )
   (batchnorm2): BatchNorm3d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
 )
 (conv): Conv2d(16, 3, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
 (lstm_cell_layer_1): LSTMCell(1662, 512)
 (lstm_cell_layer_2): LSTMCell(512, 512)
 (fc): Linear(in_features=12800, out_features=11, bias=True)
)
```

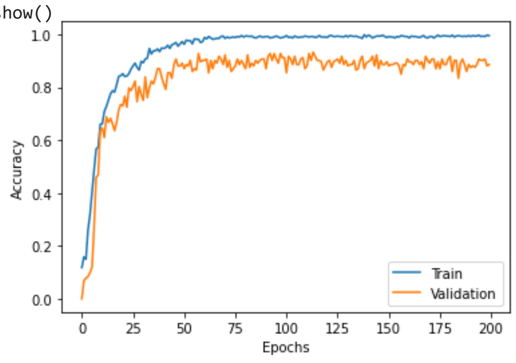
```
EPOCHS=200
trainer = Trainer(epochs=EPOCHS)
history = trainer.fit(train_dataloader, test_dataloader, model)
```

```
<---- Epoch 1: 0m -27s ---->
Train Loss: 8.746479776170519, Train Accuracy: 0.11805555555555555
Val Loss: 2.4667880535125732, Val Accuracy: 0.0
<---- Epoch 2: 0m -25s ---->
Train Loss: 6.584973547193739, Train Accuracy: 0.1579861111111111
Val Loss: 2.517930269241333, Val Accuracy: 0.06770833333333333
<---- Epoch 3: 0m -25s ---->
Train Loss: 5.780594613817003, Train Accuracy: 0.14930555555555555
Val Loss: 2.556313673655192, Val Accuracy: 0.078125
<---- Epoch 4: 0m -25s ---->
Train Loss: 4.137155479855007, Train Accuracy: 0.2586805555555556
Val Loss: 2.5263237953186035, Val Accuracy: 0.08333333333333333
<---- Epoch 5: 0m -25s ---->
Train Loss: 3.1525993082258434, Train Accuracy: 0.3194444444444444
Val Loss: 2.482621351877848, Val Accuracy: 0.09895833333333333
<---- Epoch 6: 0m -25s ---->
Train Loss: 2.3362210194269815, Train Accuracy: 0.4045138888888889
Val Loss: 2.383608102798462, Val Accuracy: 0.11979166666666667
<---- Epoch 7: 0m -25s ---->
Train Loss: 1.9143384959962633, Train Accuracy: 0.4878472222222222
Val Loss: 1.9587808847427368, Val Accuracy: 0.2708333333333333
<---- Epoch 8: 0m -25s ---->
Train Loss: 1.4917008876800537, Train Accuracy: 0.5659722222222222
Val Loss: 1.621314565340678, Val Accuracy: 0.4583333333333333
<---- Epoch 9: 0m -25s ---->
Train Loss: 1.3395700057347615, Train Accuracy: 0.5729166666666666
Val Loss: 1.4103127320607503, Val Accuracy: 0.46875
<---- Epoch 10: 0m -25s ---->
Train Loss: 1.1406449675559998, Train Accuracy: 0.6597222222222222
Val Loss: 1.1817770600318909, Val Accuracy: 0.6354166666666666
<---- Epoch 11: 0m -25s ---->
Train Loss: 1.0344972809155781, Train Accuracy: 0.6614583333333334
Val Loss: 1.1598804791768391, Val Accuracy: 0.6458333333333334
<---- Epoch 12: 0m -25s ---->
Train Loss: 0.919463402695126, Train Accuracy: 0.7083333333333334
Val Loss: 1.1914299726486206, Val Accuracy: 0.609375
<---- Epoch 13: 0m -25s ---->
Train Loss: 0.847939689954122, Train Accuracy: 0.7274305555555556
Val Loss: 1.0189509391784668, Val Accuracy: 0.6875
<---- Epoch 14: 0m -25s ---->
Train Loss: 0.7595741285218133, Train Accuracy: 0.7517361111111112
Val Loss: 1.0703515807787578, Val Accuracy: 0.6666666666666666
<---- Epoch 15: 0m -25s ---->
Train Loss: 0.6890788806809319, Train Accuracy: 0.7743055555555556
Val Loss: 1.0902127226193745, Val Accuracy: 0.6822916666666666
<---- Epoch 16: 0m -25s ---->
Train Loss: 0.703280485338635, Train Accuracy: 0.7864583333333334
Val Loss: 0.9523678024609884, Val Accuracy: 0.6614583333333334
<---- Epoch 17: 0m -25s ---->
Train Loss: 0.6142426696088579, Train Accuracy: 0.78125
Val Loss: 0.9994736313819885, Val Accuracy: 0.6354166666666666
<---- Epoch 18: 0m -25s ---->
Train Loss: 0.5637573997179667, Train Accuracy: 0.8090277777777778
Val Loss: 0.9186229308446249, Val Accuracy: 0.6666666666666666
<---- Epoch 19: 0m -25s ---->
Train Loss: 0.49826212061776054, Train Accuracy: 0.8402777777777778
Val Loss: 0.8775711258252462, Val Accuracy: 0.7135416666666666
<---- Epoch 20: 0m -25s ---->
```

```
g1 = sns.lineplot(x=range(0, EPOCHS), y=history["train"]["loss'
g2 = sns.lineplot(x=range(0, EPOCHS), y=history["val"]["loss"])
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend(["Train", "Validation"])
plt.show()
```



```
g1 = sns.lineplot(x=range(0, EPOCHS), y=history["train"]["accuracy'
g2 = sns.lineplot(x=range(0, EPOCHS), y=history["val"]["accuracy"])
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend(["Train", "Validation"])
plt.show()
```

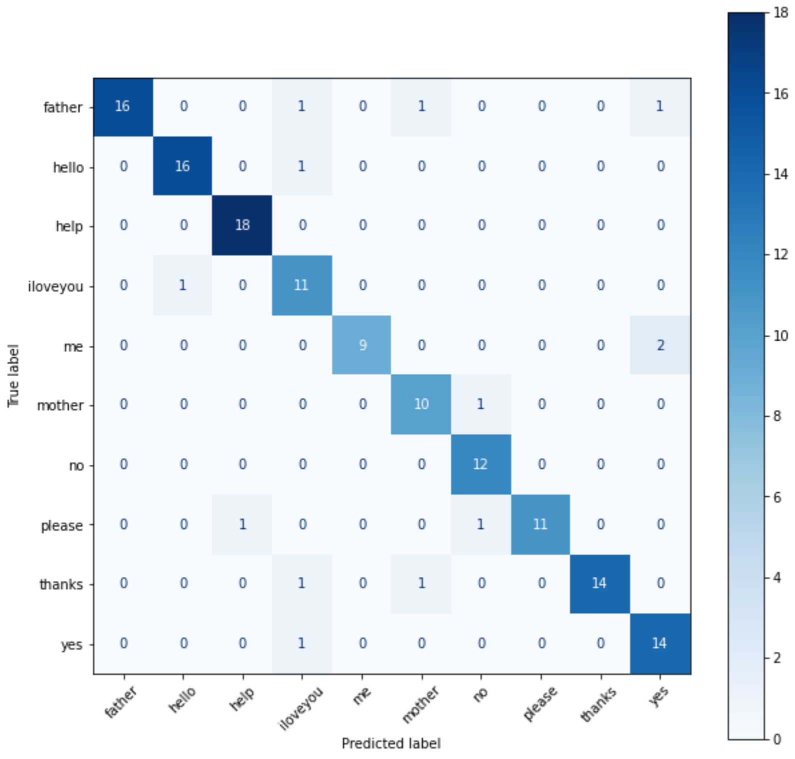


```
trainer = Trainer(epochs=EPOCHS)
trainer.load_checkpoint(epoch=114)
```

```
acc, y_pred, y = trainer.evaluate(test_dataload
print(f"Test Accuracy: {acc}")
```

```
    Test Accuracy: 0.901041666666
```

```
fig = plt.figure(figsize = (10, 10))
ax = fig.add_subplot(1, 1, 1)
cm = confusion_matrix(y.astype(int), y_pred.astype(int))
cm = ConfusionMatrixDisplay(cm, display_labels = CLASSES)
cm.plot(values_format = 'd', cmap = 'Blues', ax = ax)
plt.xticks(rotation = 45)
plt.show()
```



True label \ Predicted label	father	hello	help	iloveyou	me	mother	no	please	thanks	yes
father	16	0	0	1	0	1	0	0	0	1
hello	0	16	0	1	0	0	0	0	0	0
help	0	0	18	0	0	0	0	0	0	0
iloveyou	0	1	0	11	0	0	0	0	0	0
me	0	0	0	0	9	0	0	0	0	2
mother	0	0	0	0	0	10	1	0	0	0
no	0	0	0	0	0	0	12	0	0	0
please	0	0	1	0	0	0	1	11	0	0
thanks	0	0	0	1	0	1	0	0	14	0
yes	0	0	0	1	0	0	0	0	0	14

Check Runtime

```
gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
if gpu_info.find('failed') >= 0:
  print('Not connected to a GPU')
else:
  print(gpu_info)
```

```
    Thu Dec 15 03:07:40 2022
    +-----------------------------------------------------------------------------+
    | NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
    |-------------------------------+----------------------+----------------------+
    | GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
    | Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
    |                               |                      |               MIG M. |
    |===============================+======================+======================|
    |   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
    | N/A   44C    P0    26W /  70W |      0MiB / 15109MiB |      0%      Default |
    |                               |                      |                  N/A |
    +-------------------------------+----------------------+----------------------+

    +-----------------------------------------------------------------------------+
    | Processes:                                                                  |
    |  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
    |        ID   ID                                                   Usage      |
    |=============================================================================|
    |  No running processes found                                                 |
    +-----------------------------------------------------------------------------+
```

```
from psutil import virtual_memory
ram_gb = virtual_memory().total / 1e9
print('Your runtime has {:.1f} gigabytes of available RAM\n'.format(ram_gb))

if ram_gb < 20:
  print('Not using a high-RAM runtime')
else:
  print('You are using a high-RAM runtime!')
```

```
    Your runtime has 27.3 gigabytes of available RAM

    You are using a high-RAM runtime!
```

Import Libraries

```
import os
import cv2
import matplotlib.pyplot as plt
import numpy as np
from datetime import datetime
import random
import time
import json
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, ConfusionMatrixDisplay


import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from torchvision.utils import make_grid
import torch.nn.functional as F
from torch.utils.data import Subset, Dataset
import torch.optim.lr_scheduler as lr_scheduler
from torchsummary import summary
```

Global Variables

```
SEED = 42

np.random.seed(SEED)
```

```
random.seed(SEED)

BATCH_SIZE=64

LEARNING_RATE=0.0001
WEIGHT_DECAY=0.8
# SGD
MOMENTUM=0.9

IMAGE_HEIGHT , IMAGE_WIDTH = 64, 64

FRAMES_PER_VIDEO = 20

ROOT_DIR=os.path.join('drive', 'MyDrive', 'UCF-11')
DEVICE = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
# DEVICE = "cpu"
```

Dataset

```
from google.colab import dr
drive.mount('/content/drive')
```

```
     Mounted at /content/dr
```

```
X_train_frames = np.load(os.path.join(ROOT_DIR, "Train", "X_train_frames.npy"))
X_train_keypoints = np.load(os.path.join(ROOT_DIR, "Train", "X_train_keypoints.npy"))
y_train = np.load(os.path.join(ROOT_DIR, "Train", "y_train.npy"))

X_test_frames = np.load(os.path.join(ROOT_DIR, "Test", "X_test_frames.npy"))
X_test_keypoints = np.load(os.path.join(ROOT_DIR, "Test", "X_test_keypoints.npy"))
y_test = np.load(os.path.join(ROOT_DIR, "Test", "y_test.npy"))

X_val_frames = np.load(os.path.join(ROOT_DIR, "Val", "X_val_frames.npy"))
X_val_keypoints = np.load(os.path.join(ROOT_DIR, "Val", "X_val_keypoints.npy"))
y_val = np.load(os.path.join(ROOT_DIR, "Val", "y_val.npy"))
```

```
with open(os.path.join(ROOT_DIR, "label_encoder.json")) as
    data = json.load(f)
CLASSES = list(data.keys())
```

```
class UCFCustom(Dataset):
    def __init__(self, frames, keypoints, labels, transform=None):
        self.frames = frames
        self.labels = labels
        self.keypoints = keypoints
        self.transform = transform

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, index):
        X_frames = np.moveaxis(self.frames[index], 3, 0)
        X_frames = torch.from_numpy(X_frames)
        X_keypoints = torch.from_numpy(self.keypoints[index])
        y = torch.from_numpy(np.array(self.labels[index])).type(torch.LongTensor)

        X_frames = X_frames.float()
        X_keypoints = X_keypoints.float()
        # y = y.float()


        if self.transform:
            X_frames = self.transform(X_frames)
            X_keypoints = self.transform(X_keypoints)

        return X_frames, X_keypoints, y
```

```
test_dataset = UCFCustom(X_test_frames, X_test_keypoints, y_test, transform=None)
val_dataset = UCFCustom(X_val_frames, X_val_keypoints, y_val, transform=None)
train_dataset = UCFCustom(X_train_frames, X_train_keypoints, y_train, transform=None)
```

```
test_dataloader = DataLoader(test_dataset, batch_size=BATCH_SIZE,
                                          shuffle=True, num_workers=0)
train_dataloader = DataLoader(train_dataset, batch_size=BATCH_SIZE,
                                          shuffle=True, num_workers=0)
```

```
val_dataloader = DataLoader(val_dataset, batch_size=BATCH_SIZE,
                                          shuffle=True, num_workers=0)
```

```
X_frames, X_keypoints, y = next(iter(train_dataloade
```

```
X_frames.sh
```

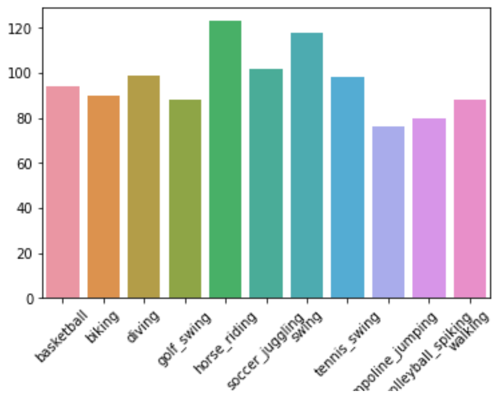
```
    torch.Size([64, 3, 20, 64, 6
```

```
X_keypoints.sh
```

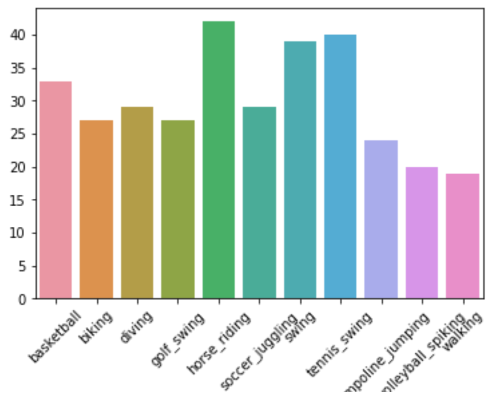
```
    torch.Size([64, 20, 166
```

```
sample = dic
```

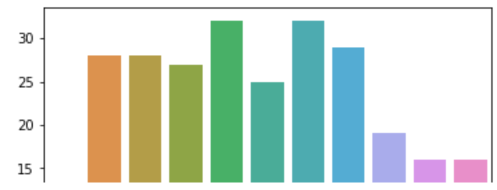
```
x_axis, y_axis = np.unique(y_train, return_counts=Tr
sns.barplot(x=CLASSES, y=y_axis)
plt.xticks(rotation=45)
plt.show()
sample["train"] = list(y_axis)
```



```
x_axis, y_axis = np.unique(y_val, return_counts=Tr
sns.barplot(x=CLASSES, y=y_axis)
plt.xticks(rotation=45)
plt.show()
sample["val"] = list(y_axis)
```



```
x_axis, y_axis = np.unique(y_test, return_counts=Tr
sns.barplot(x=CLASSES, y=y_axis)
plt.xticks(rotation=45)
plt.show()
sample["test"] = list(y_axis)
```



```
import pandas as pd

df = pd.DataFrame(data=sample, index=CLASSES)
```

```
from google.colab import fi
df.to_csv('filename.csv')
files.download('filename.csv')
```

Model

Module

[ ] ↳ *1 cell hidden*

ConvLSTM

[ ] ↳ *2 cells hidden*

Architecture

[ ] ↳ *1 cell hidden*

TRAINER

[ ] ↳ *1 cell hidden*

TRAINING

```
# The input video frames are grayscale, thus single channel
model = Seq2Seq(num_channels=3, num_kernels=16,
kernel_size=(5, 5), padding=(2, 2), activation="relu",
frame_size=(64, 64), num_layers=2, sequence_length=20, keypoints_length=1662).to(DEVICE)

def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

count_parameters(model)
```

```
     7174
```

```
print(mod
```

```
      eq2Seq(
       (sequential): Sequential(
         (convlstm1): ConvLSTM(
           (convLSTMcell): ConvLSTMCell(
             (conv): Conv2d(19, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
           )
         )
         (batchnorm1): BatchNorm3d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
         (convlstm2): ConvLSTM(
           (convLSTMcell): ConvLSTMCell(
             (conv): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
           )
         )
         (batchnorm2): BatchNorm3d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
       )
       (conv): Conv2d(16, 3, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
       (lstm_cell_layer_1): LSTMCell(1662, 512)
       (lstm_cell_layer_2): LSTMCell(512, 512)
```

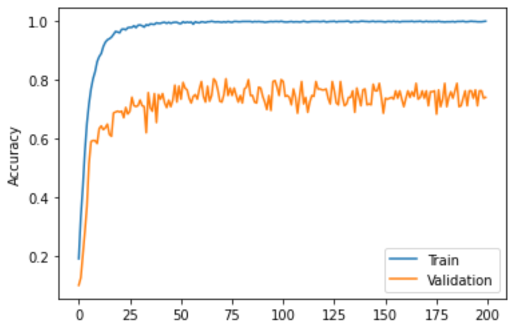
```
    (fc): Linear(in_features=12800, out_features=11, bias=True)
  )
```

```
EPOCHS=200
trainer = Trainer(epochs=EPOCHS)
history = trainer.fit(train_dataloader, test_dataloader, model)
```

```
<---- Epoch 1: 0m -34s ---->
Train Loss: 4.794933795928955, Train Accuracy: 0.19025735294117646
Val Loss: 2.41970009803772, Val Accuracy: 0.1
<---- Epoch 2: 0m -27s ---->
Train Loss: 2.6868588784161735, Train Accuracy: 0.3318014705882353
Val Loss: 2.410547161102295, Val Accuracy: 0.12544642857142857
<---- Epoch 3: 0m -27s ---->
Train Loss: 1.999521304579342, Train Accuracy: 0.43474264705882354
Val Loss: 2.3567006587982178, Val Accuracy: 0.20357142857142857
<---- Epoch 4: 0m -27s ---->
Train Loss: 1.4446458115297205, Train Accuracy: 0.5560661764705882
Val Loss: 2.2826026678085327, Val Accuracy: 0.28214285714285714
<---- Epoch 5: 0m -27s ---->
Train Loss: 1.0945818213855518, Train Accuracy: 0.6516544117647058
Val Loss: 2.0156840085983276, Val Accuracy: 0.36919642857142854
<---- Epoch 6: 0m -27s ---->
Train Loss: 0.8878033196224886, Train Accuracy: 0.7178308823529411
Val Loss: 1.8413636922836303, Val Accuracy: 0.5138392857142857
<---- Epoch 7: 0m -27s ---->
Train Loss: 0.7630745102377499, Train Accuracy: 0.7683823529411765
Val Loss: 1.5529345989227294, Val Accuracy: 0.5897321428571429
<---- Epoch 8: 0m -27s ---->
Train Loss: 0.688280280898599, Train Accuracy: 0.8033088235294118
Val Loss: 1.7031830549240112, Val Accuracy: 0.5928571428571429
<---- Epoch 9: 0m -27s ---->
Train Loss: 0.6169318802216474, Train Accuracy: 0.8272058823529411
Val Loss: 1.492460584640503, Val Accuracy: 0.5924107142857142
<---- Epoch 10: 0m -27s ---->
Train Loss: 0.5424344276680666, Train Accuracy: 0.8612132352941176
Val Loss: 1.5116236448287963, Val Accuracy: 0.5825892857142857
<---- Epoch 11: 0m -27s ---->
Train Loss: 0.4850982076981488, Train Accuracy: 0.8786764705882353
Val Loss: 1.4195921182632447, Val Accuracy: 0.6334821428571429
<---- Epoch 12: 0m -27s ---->
Train Loss: 0.45635365913896, Train Accuracy: 0.8887867647058824
Val Loss: 1.432945227622986, Val Accuracy: 0.6424107142857143
<---- Epoch 13: 0m -27s ---->
Train Loss: 0.408099163981045, Train Accuracy: 0.9126838235294118
Val Loss: 1.3642434358596802, Val Accuracy: 0.6299107142857142
<---- Epoch 14: 0m -27s ---->
Train Loss: 0.38012513343025656, Train Accuracy: 0.9273897058823529
Val Loss: 1.3128852128982544, Val Accuracy: 0.6361607142857142
<---- Epoch 15: 0m -27s ---->
Train Loss: 0.34252826431218314, Train Accuracy: 0.9356617647058824
Val Loss: 1.4722770214080811, Val Accuracy: 0.6486607142857143
<---- Epoch 16: 0m -27s ---->
Train Loss: 0.3252798424047582, Train Accuracy: 0.9393382352941176
Val Loss: 1.48217613697052, Val Accuracy: 0.6138392857142857
<---- Epoch 17: 0m -27s ---->
Train Loss: 0.30680298717582927, Train Accuracy: 0.9448529411764706
Val Loss: 1.6009910106658936, Val Accuracy: 0.6071428571428571
<---- Epoch 18: 0m -27s ---->
Train Loss: 0.2885063877877067, Train Accuracy: 0.9540441176470589
Val Loss: 1.2053990125656129, Val Accuracy: 0.6866071428571429
<---- Epoch 19: 0m -27s ---->
Train Loss: 0.2676975122269462, Train Accuracy: 0.9641544117647058
Val Loss: 1.1658024191856384, Val Accuracy: 0.6897321428571429
<---- Epoch 20: 0m -27s ---->
```

```
g1 = sns.lineplot(x=range(0, 200), y=history["train"]["loss'
g2 = sns.lineplot(x=range(0, 200), y=history["val"]["loss"])
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend(["Train", "Validation"])
plt.show()
```

```
g1 = sns.lineplot(x=range(0, 200), y=history["train"]["accuracy'
g2 = sns.lineplot(x=range(0, 200), y=history["val"]["accuracy"])
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend(["Train", "Validation"])
plt.show()
```

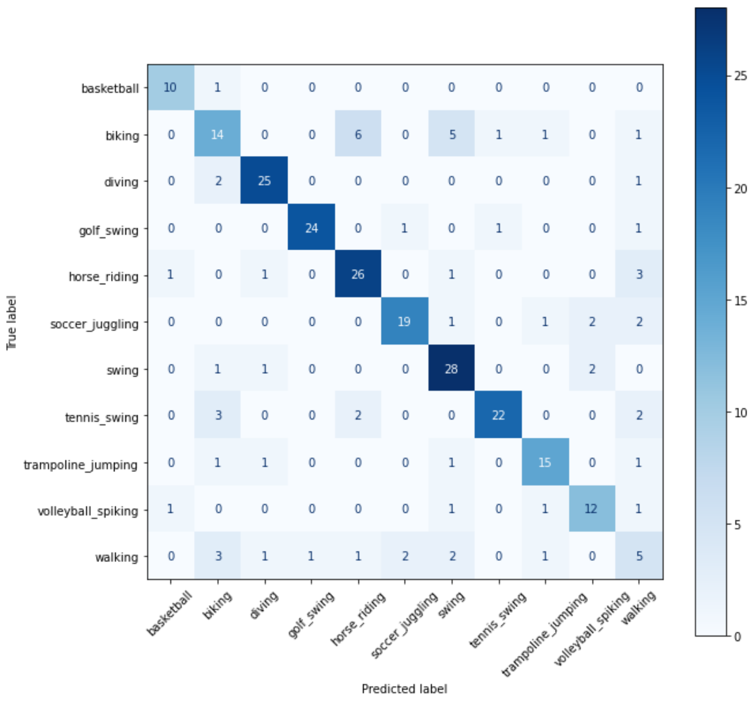


```
trainer = Trainer(epochs=EPOCHS
trainer.load_checkpoint(epoch=73)
```

```
acc, y_pred, y = trainer.evaluate(test_dataloade
print(f"Test Accuracy: {acc}")
```

```
    Test Accuracy: 0.7776785714285
```

```
fig = plt.figure(figsize = (10, 10))
ax = fig.add_subplot(1, 1, 1)
cm = confusion_matrix(y.astype(int), y_pred.astype(int))
cm = ConfusionMatrixDisplay(cm, display_labels = CLASSES)
cm.plot(values_format = 'd', cmap = 'Blues', ax = ax)
plt.xticks(rotation = 45)
plt.show()
```


basketball 10 1 0 0 0 0 0 0 0 0 0
biking 0 14 0 0 6 0 5 1 1 0 1
diving 0 2 25 0 0 0 0 0 0 0 1
golf_swing 0 0 0 24 0 1 0 1 0 0 1
horse_riding 1 0 1 0 26 0 1 0 0 0 3
soccer_juggling 0 0 0 0 0 19 1 0 1 2 2
swing 0 1 1 0 0 0 28 0 0 2 0
tennis_swing 0 3 0 0 2 0 0 22 0 0 2
trampoline_jumping 0 1 1 0 0 0 1 0 15 0 1
volleyball_spiking 1 0 0 0 0 0 1 0 1 12 1
walking 0 3 1 1 1 2 2 0 1 0 5
True label
basketball
biking
diving
golf_swing
horse_riding
soccer_juggling
swing
tennis_swing
trampoline_jumping
volleyball_spiking
walking
Predicted label