INFOMCV 2017/2018

Assignment 4 Report – Action Classification in Video

Stephan Wells
Game & Media Technology
Graduate School of Natural Sciences
Utrecht University
Utrecht, Netherlands
s.b.wells@students.uu.nl
6157114

Erik Scerri
Game & Media Technology
Graduate School of Natural Sciences
Utrecht University
Utrecht, Netherlands
e.scerri@students.uu.nl
6158811

OVERVIEW

Objectives Met:

- **Training a CNN (10)**
- **Testing a video on a single frame (10)**
- **K-Fold Cross Validation (10)**
- **Performance Measurement (10)**
- **Train on mirrored frames (5)**
- **Test performance using different sets of parameters (10)**
- **Test performance on different network topologies (5)**
- **Test performance on your test sets and own recordings (20)**
- **Test performance on all frames of a test video (5)**

References:

- Lecture slides.
- OpenCV documentation: https://docs.opencv.org/master/
- TensorFlow documentation: https://www.tensorflow.org/api_docs/
- 3Blue1Brown's neural network explanations: https://www.youtube.com/watch?v=aircAruvnKk

Division of Work:

- Stephan Wells:
  - Coded functionality for confusion matrices.
  - Implemented cross validation.
  - Worked on tuning parameters.
  - Wrote the report.
  - Performed general research & debugging.
  - Recorded videos for test data.
- Erik Scerri:
  - Handled file I/O and processing.
  - Set up the convolutional neural network.
  - Worked on tuning parameters.
  - Wrote the report.
  - Performed general research & debugging.
  - Recorded videos for test data.

IMPLEMENTATION

The purpose of this section is to detail the implemented functionality through explanations and code snippers and to give an overview of the training/testing pipeline developed for the assignment.

The project was built in Python 3.6, using OpenCV and TensorFlow. Work was done on the following files:

- main.py: Contains the code that processes video frames and calls training/evaluation methods.
- cnn.py: Contains the TensorFlow setup of the CNN, including the layer topology.
- validation.py: Contains code that performs cross validation and generates performance measures.

Data Loading/Processing:

Loading of video frames is done through OpenCV in Python. The default method is to load the middle frame from the video, using `getFrameMat(path)`. Functionality to load a specific frame (or multiple frames) from a video was also added.

```python
def getFrameMat(path):
    vid = cv.VideoCapture(path)
    count = vid.get(cv.CAP_PROP_FRAME_COUNT)
    vid.set(cv.CAP_PROP_POS_FRAMES, count / 2)
    _, frame = vid.read()
    vid.release()
    return frame
```

The main data loading method, `getFrameMats(basepath)` repeatedly calls the above methods for all files in the given base directory. The `basepath` parameter is expected to point to a folder which contains 5 subfolders, one per Action. Loaded frames are **reshaped to a standard 90x90** image dimension. Note that this method is also where **loaded frames are horizontally flipped** to serve as an augmentation to the training data. This method returns two lists, one is the list of frames (90x90x3 matrices), and one is the list of labels corresponding to each image (their class.)

```python
        for filename in os.listdir(datapath):
            frame = getFrameMat(datapath + filename)
            mat = cv.resize(frame, (90, 90))

            mat_orig = np.divide(np.float32(mat), 255)
            labels.append(action.value)
            mats.append(mat_orig)

            mat_flip = cv.flip(mat_orig, 1)
            labels.append(action.value)
            mats.append(mat_flip)
```

Cross Validation:

`validation.py` deals with all of the code related to cross validation, evaluation, and performance measures. k-fold cross validation was handled by the `crossValidation()` method, which takes a single parameter (*k*) and splits the training data in a uniformly random manner into *k* folds. It then runs through *k* iterations, where a different fold is used for validation each time and the rest of the folds are concatenated for the training data.

The below code snippet from the aforementioned method shows how the fold segregation functions.

```python
end_offset = 1 if i < (k - 1) else 0
left_data = data[0:fold_size * i]
left_labels = labels[0:fold_size * i]
right_data = data[fold_size * (i + 1) + end_offset:]
right_labels = labels[fold_size * (i + 1) + end_offset:]
```

Performance Measures:

Four performance measures were calculated: precision, recall, F-score, and confusion matrices. The respective methods for these performance measures are all in `validation.py`. For each fold iteration, the iteration's confusion matrix is calculated, and its values are added to a cumulative confusion matrix in the following code snippet found in `crossValidation()`.

```
predicted_labels = []

for result in results:
    predicted_labels.append(result['classes'])

conf_mat = generateConfusionMatrix(eval_labels, predicted_labels)
overall_conf_mat = np.add(overall_conf_mat, conf_mat)
```

This overall confusion matrix is then averaged after all of the fold iterations are carried out. It is outputted on screen and used to calculate the of the performance measures and output them to file.

```
averaged_conf_mat = np.divide(np.float32(overall_conf_mat), k)
outputConfusionMatrix(averaged_conf_mat)
generatePerfMeasures(averaged_conf_mat)
```

The Neural Network:

The network topology is defined and constructed inside `cnn.py`, specifically in the method `cnn_model()`, which accepts a list of inputs and a list of expected labels, and then defines how these will traverse the network. The topology is defined using standard Tensorflow instructions, notably `conv2d()`, `max_pooling2d()`, and `dense()`.

```
inputLayer = tf.reshape(features["x"], [-1, 90, 90, 3])

conv1 = tf.layers.conv2d(
        inputs=inputLayer,
        filters=32,
        kernel_size=[7, 7],
        padding="valid",
        activation=tf.nn.relu)

pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2], strides=2)
```

Important parameters such as **Learning Rate** and **Dropout Rate** are also defined in this file. The model is called for training, evaluation, or prediction, and will return different outputs for either. Notably, for evaluation mode it will return a general accuracy and loss value, and for prediction mode it will return a list of predicted classes.

```
predictions = {
 "classes": tf.argmax(input=logits, axis=1),
 "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
}

if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)
```

RESULTS

The initial tests being run here are designed to determine an optimal layout for the neural network. As such, all parameter tuning is done by testing on a validation set using cross validation. This means that accuracy results here are not representative of actual performance on a proper test set but do serve as a guide for improvement in terms of parameter tuning. **All results given here are averages over all folds.**

Baseline Test:

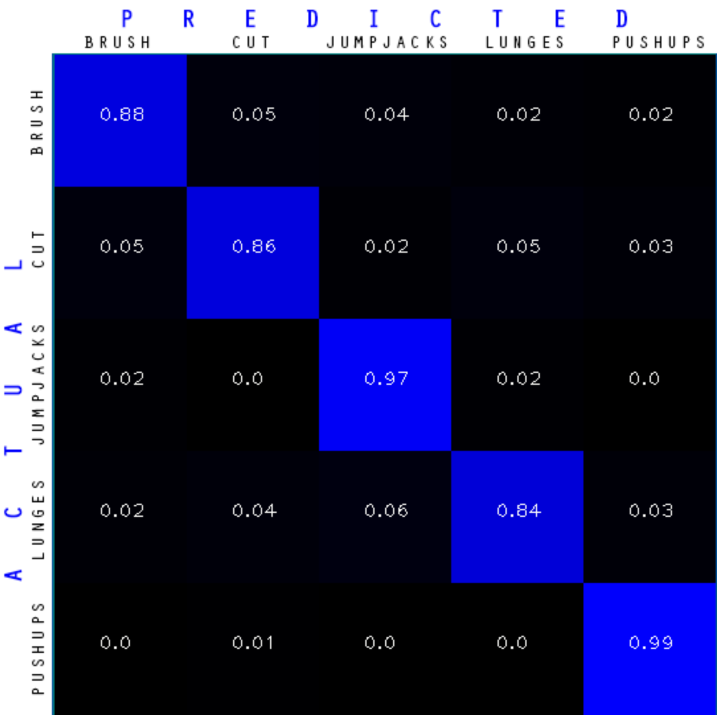
testA1: Baseline	
Learning Rate	0.001
Dropout Rate	0.3
Batch Size	10
Num of Epochs	10000

Class	**0**	**1**	**2**	**3**	**4**
Precision	0.878	0.862	0.967	0.843	0.992
Recall	0.911	0.903	0.890	0.908	0.930
F-Score	0.894	0.882	0.927	0.874	0.960

Average Precision	0.9084
Standard Dev.	0.0596

Average Recall	0.9084
Standard Dev.	0.0130

Average F-Score	0.9074
Standard Dev.	0.0319



ACTUAL \ PREDICTED	BRUSH	CUT	JUMPJACKS	LUNGES	PUSHUPS
BRUSH	0.88	0.05	0.04	0.02	0.02
CUT	0.05	0.86	0.02	0.05	0.03
JUMPJACKS	0.02	0.0	0.97	0.02	0.0
LUNGES	0.02	0.04	0.06	0.84	0.03
PUSHUPS	0.0	0.01	0.0	0.0	0.99

Train on Mirrored Frames:

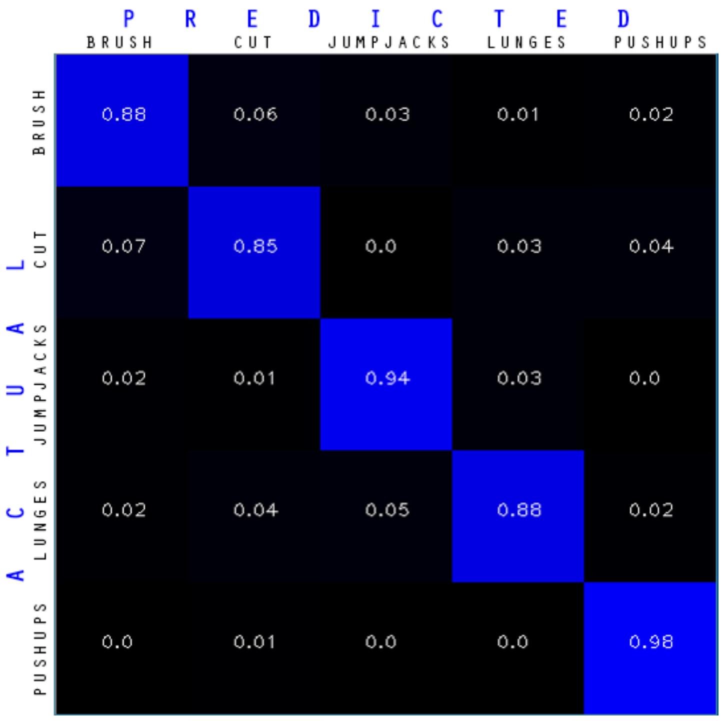
testA2: Augmented Data	
Learning Rate	0.001
Dropout Rate	0.3
Batch Size	10
Num of Epochs	10000

Class	**0**	**1**	**2**	**3**	**4**
Precision	0.877	0.855	0.939	0.877	0.981
Recall	0.889	0.873	0.916	0.917	0.931
F-Score	0.884	0.863	0.928	0.901	0.956

Average Precision	0.9058
Standard Dev.	0.0469

Average Recall	0.9052
Standard Dev.	0.0210

Average F-Score	0.9074
Standard Dev.	0.0327



ACTUAL \ PREDICTED	BRUSH	CUT	JUMPJACKS	LUNGES	PUSHUPS
BRUSH	0.88	0.06	0.03	0.01	0.02
CUT	0.07	0.85	0.0	0.03	0.04
JUMPJACKS	0.02	0.01	0.94	0.03	0.0
LUNGES	0.02	0.04	0.05	0.88	0.02
PUSHUPS	0.0	0.01	0.0	0.0	0.98

Parameter Tuning – Batch Size:

Each parameter for tuning is tested with five different values. The best of the five values is carried over to the next test. For Batch Size, the tested values are: **10, 15, 20, 25, 50**. Since Batch Size of 10 with this network setup is already tested above, the test is not repeated here.

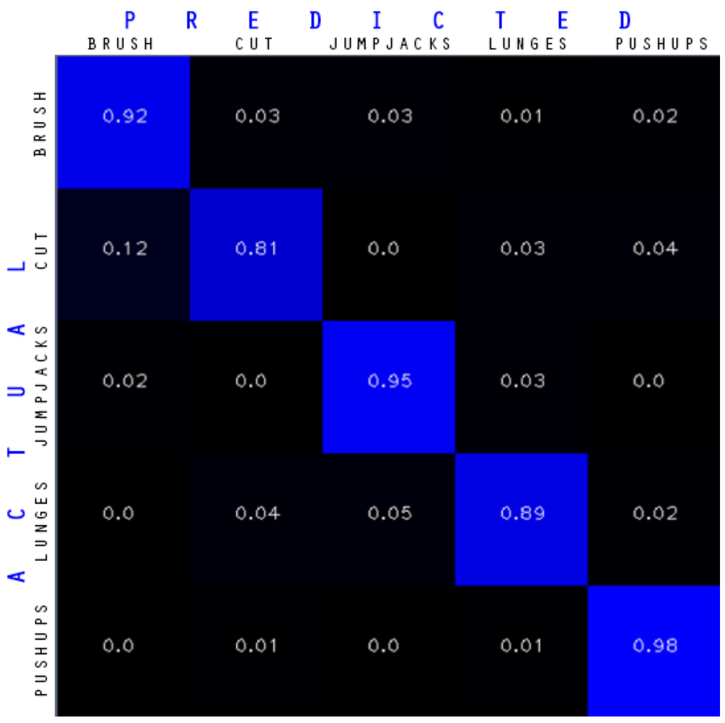
testB1: Batch Size Tuning	
Learning Rate	0.001
Dropout Rate	0.3
Batch Size	**15**
Num of Epochs	10000

Class	**0**	**1**	**2**	**3**	**4**
Precision	0.916	0.809	0.951	0.889	0.985
Recall	0.865	0.908	0.928	0.922	0.929
F-Score	0.890	0.856	0.939	0.905	0.956

Average Precision	0.9100
Standard Dev.	0.0601

Average Recall	0.9104
Standard Dev.	0.0239

Average F-Score	0.9092
Standard Dev.	0.0355



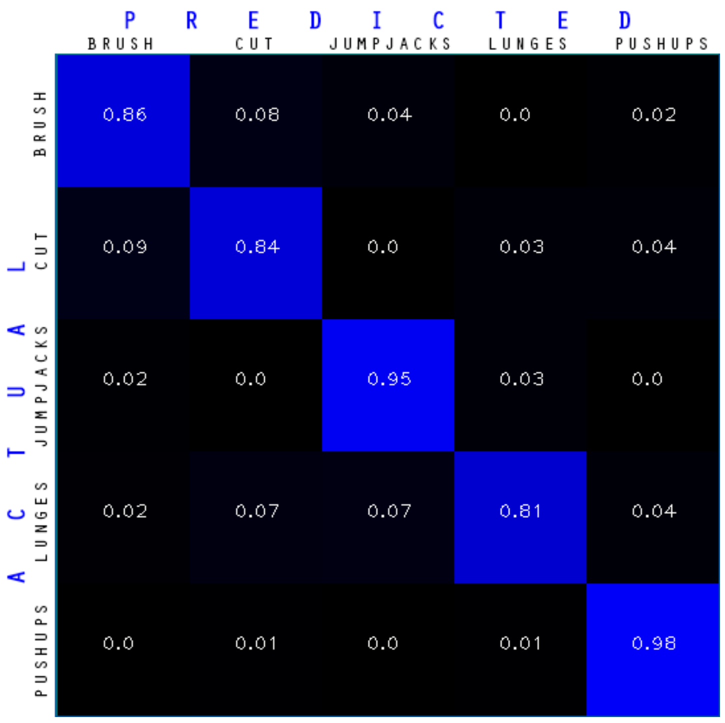
testB2: Batch Size Tuning	
Learning Rate	0.001
Dropout Rate	0.3
Batch Size	**20**
Num of Epochs	10000

Class	**0**	**1**	**2**	**3**	**4**
Precision	0.859	0.841	0.951	0.807	0.981
Recall	0.871	0.841	0.897	0.910	0.918
F-Score	0.865	0.841	0.923	0.855	0.949

Average Precision	0.8878
Standard Dev.	0.0667

Average Recall	0.8874
Standard Dev.	0.0281

Average F-Score	0.8866
Standard Dev.	0.0419



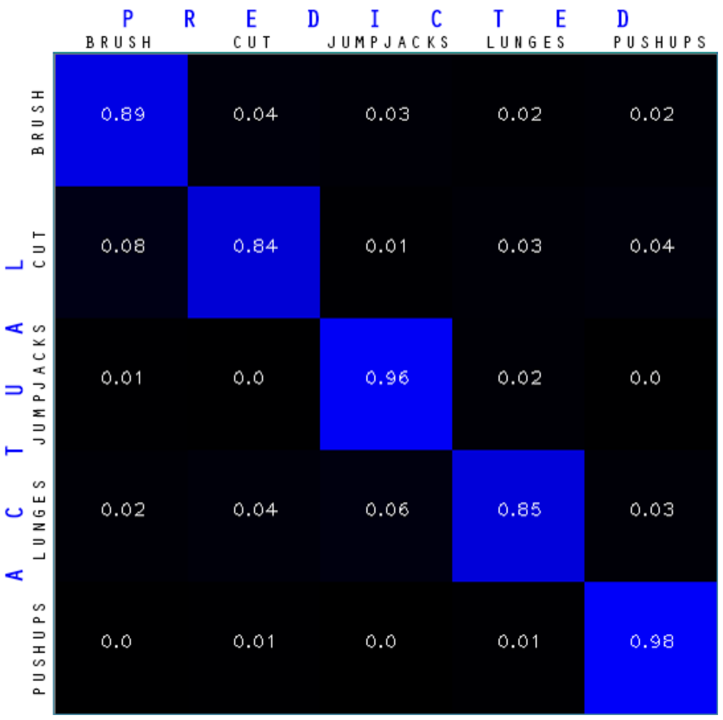
testB3: Batch Size Tuning	
Learning Rate	0.001
Dropout Rate	0.3
Batch Size	**25**
Num of Epochs	10000

Class	**0**	**1**	**2**	**3**	**4**
Precision	0.893	0.836	0.959	0.850	0.977
Recall	0.884	0.899	0.906	0.915	0.911
F-Score	0.888	0.867	0.932	0.881	0.943

Average Precision	0.9030
Standard Dev.	0.0566

Average Recall	0.9030
Standard Dev.	0.0109

Average F-Score	0.9022
Standard Dev.	0.0298



		PREDICTED				
		BRUSH	CUT	JUMPJACKS	LUNGES	PUSHUPS
ACTUAL	BRUSH	0.89	0.04	0.03	0.02	0.02
	CUT	0.08	0.84	0.01	0.03	0.04
	JUMPJACKS	0.01	0.0	0.96	0.02	0.0
	LUNGES	0.02	0.04	0.06	0.85	0.03
	PUSHUPS	0.0	0.01	0.0	0.01	0.98

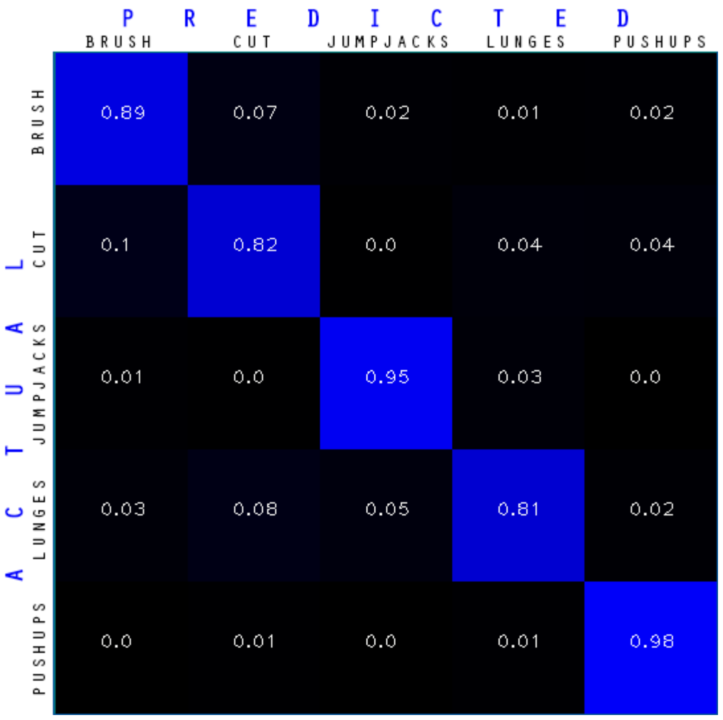
testB4: Batch Size Tuning	
Learning Rate	0.001
Dropout Rate	0.3
Batch Size	**50**
Num of Epochs	10000

Class	**0**	**1**	**2**	**3**	**4**
Precision	0.885	0.822	0.951	0.814	0.981
Recall	0.864	0.834	0.927	0.894	0.932
F-Score	0.874	0.828	0.939	0.852	0.956

Average Precision	0.8906
Standard Dev.	0.0669

Average Recall	0.8902
Standard Dev.	0.0373

Average F-Score	0.8898
Standard Dev.	0.0496



		PREDICTED				
		BRUSH	CUT	JUMPJACKS	LUNGES	PUSHUPS
ACTUAL	BRUSH	0.89	0.07	0.02	0.01	0.02
	CUT	0.1	0.82	0.0	0.04	0.04
	JUMPJACKS	0.01	0.0	0.95	0.03	0.0
	LUNGES	0.03	0.08	0.05	0.81	0.02
	PUSHUPS	0.0	0.01	0.0	0.01	0.98

Parameter Tuning – Dropout Rate:

The best Batch Size value out of the tested five is carried over to the next test. A Batch Size of 15 gave the highest average F-Score and was thus chosen to continue into the next batch of tests. For Dropout Rate values are: **0.3, 0.4, 0.5, 0.6, 0.7**. Since Dropout Rate of 0.3 with this network setup is already tested above, the test is not repeated here.

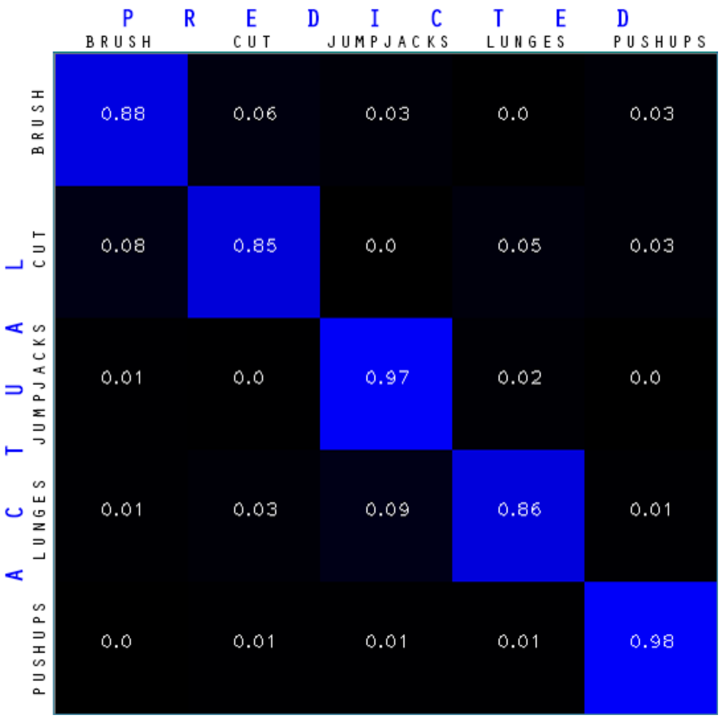
testC1: Dropout Rate Tuning	
Learning Rate	0.001
Dropout Rate	**0.4**
Batch Size	**15**
Num of Epochs	10000

Class	**0**	**1**	**2**	**3**	**4**
Precision	0.882	0.850	0.967	0.857	0.977
Recall	0.901	0.898	0.882	0.917	0.937
F-Score	0.891	0.873	0.922	0.886	0.956

Average Precision	0.9066
Standard Dev.	0.0545

Average Recall	0.9070
Standard Dev.	0.0187

Average F-Score	0.9056
Standard Dev.	0.0299



ACTUAL \ PREDICTED	BRUSH	CUT	JUMPJACKS	LUNGES	PUSHUPS
BRUSH	0.88	0.06	0.03	0.0	0.03
CUT	0.08	0.85	0.0	0.05	0.03
JUMPJACKS	0.01	0.0	0.97	0.02	0.0
LUNGES	0.01	0.03	0.09	0.86	0.01
PUSHUPS	0.0	0.01	0.01	0.01	0.98

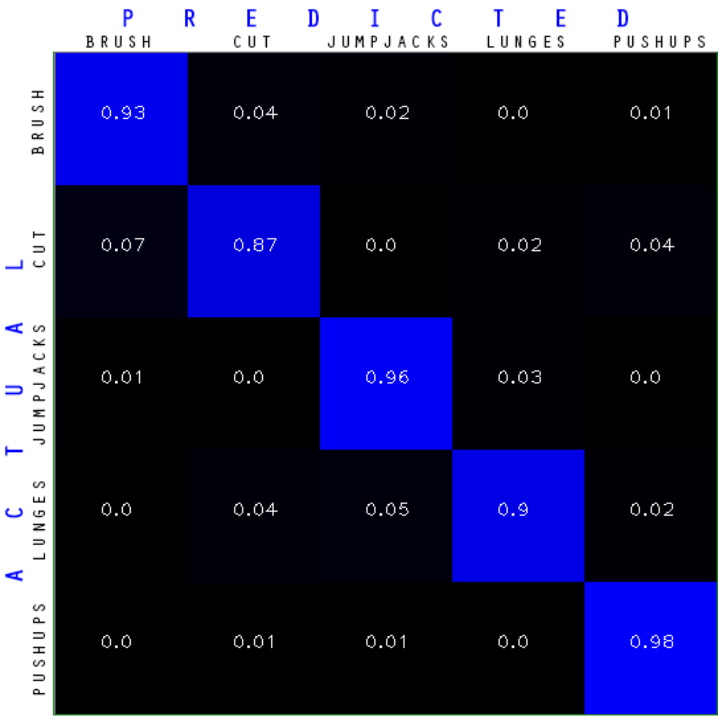
testC2: Dropout Rate Tuning	
Learning Rate	0.001
Dropout Rate	**0.5**
Batch Size	**15**
Num of Epochs	10000

Class	**0**	**1**	**2**	**3**	**4**
Precision	0.927	0.868	0.959	0.897	0.981
Recall	0.920	0.907	0.921	0.942	0.942
F-Score	0.924	0.887	0.940	0.919	0.961

Average Precision	0.9264
Standard Dev.	0.0408

Average Recall	0.9264
Standard Dev.	0.0137

Average F-Score	0.9262
Standard Dev.	0.0245



ACTUAL \ PREDICTED	BRUSH	CUT	JUMPJACKS	LUNGES	PUSHUPS
BRUSH	0.93	0.04	0.02	0.0	0.01
CUT	0.07	0.87	0.0	0.02	0.04
JUMPJACKS	0.01	0.0	0.96	0.03	0.0
LUNGES	0.0	0.04	0.05	0.9	0.02
PUSHUPS	0.0	0.01	0.01	0.0	0.98

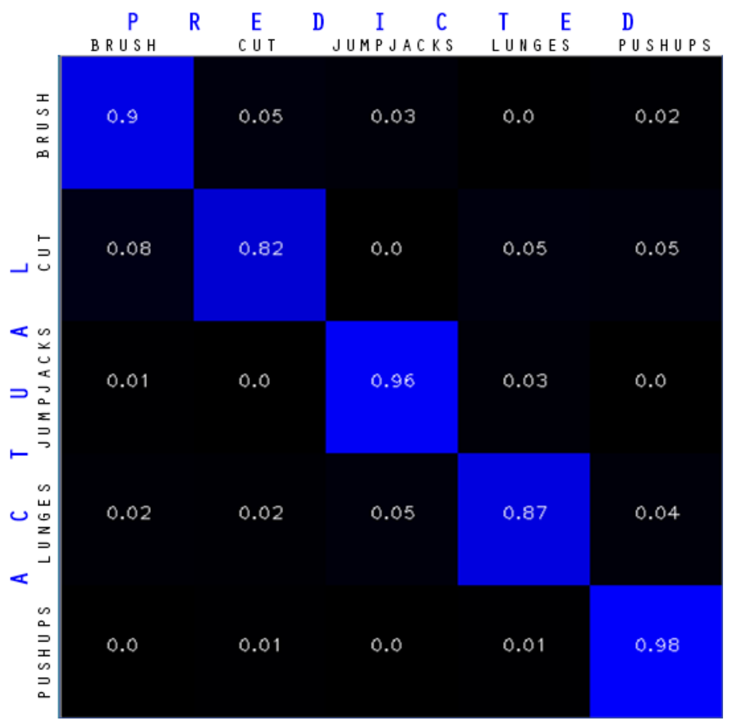
testC3: Dropout Rate Tuning	
Learning Rate	0.001
Dropout Rate	**0.6**
Batch Size	15
Num of Epochs	10000

Class	**0**	**1**	**2**	**3**	**4**
Precision	0.897	0.818	0.959	0.870	0.984
Recall	0.887	0.914	0.922	0.898	0.908
F-Score	0.892	0.863	0.940	0.884	0.945

Average Precision	0.9056
Standard Dev.	0.0600

Average Recall	0.9058
Standard Dev.	0.0122

Average F-Score	0.9048
Standard Dev.	0.0322



	BRUSH	CUT	JUMPJACKS	LUNGES	PUSHUPS
BRUSH	0.9	0.05	0.03	0.0	0.02
CUT	0.08	0.82	0.0	0.05	0.05
JUMPJACKS	0.01	0.0	0.96	0.03	0.0
LUNGES	0.02	0.02	0.05	0.87	0.04
PUSHUPS	0.0	0.01	0.0	0.01	0.98

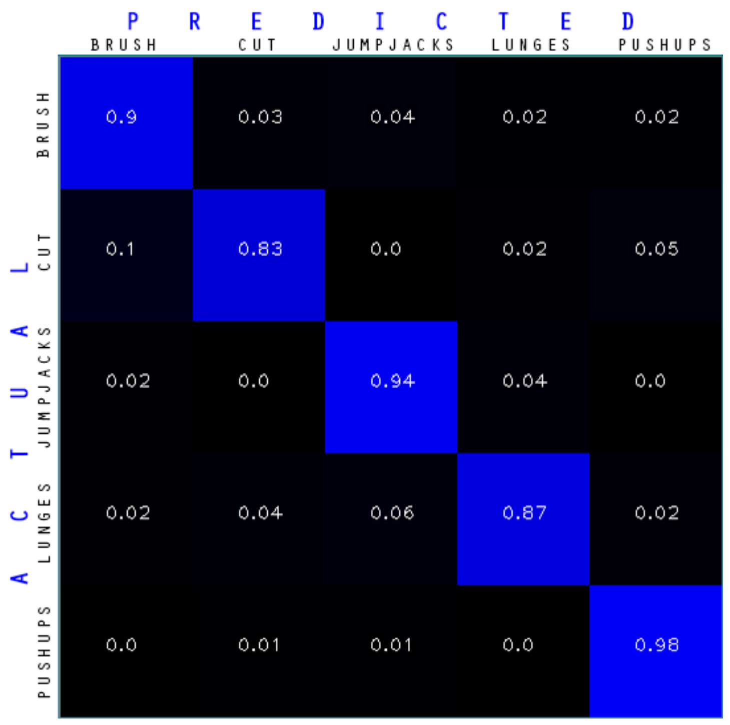
testC4: Dropout Rate Tuning	
Learning Rate	0.001
Dropout Rate	**0.7**
Batch Size	15
Num of Epochs	10000

Class	**0**	**1**	**2**	**3**	**4**
Precision	0.897	0.818	0.959	0.87	0.984
Recall	0.887	0.914	0.922	0.898	0.908
F-Score	0.892	0.863	0.94	0.884	0.945

Average Precision	0.9060
Standard Dev.	0.0490

Average Recall	0.9042
Standard Dev.	0.0220

Average F-Score	0.9044
Standard Dev.	0.0270



	BRUSH	CUT	JUMPJACKS	LUNGES	PUSHUPS
BRUSH	0.9	0.03	0.04	0.02	0.02
CUT	0.1	0.83	0.0	0.02	0.05
JUMPJACKS	0.02	0.0	0.94	0.04	0.0
LUNGES	0.02	0.04	0.06	0.87	0.02
PUSHUPS	0.0	0.01	0.01	0.0	0.98

Parameter Tuning – Learning Rate:

The best Dropout Rate value out of the tested five is carried over to the next test. A Dropout Rate of 0.5 gave the highest average F-Score and the least deviation from the average. It was therefore carried over into this next set of tests. For Learning Rate values are: **0.001, 0.0001, 0.002, 0.005, 0.05**. Since Learning Rate of 0.001 with this network setup is already tested above, the test is not repeated here.

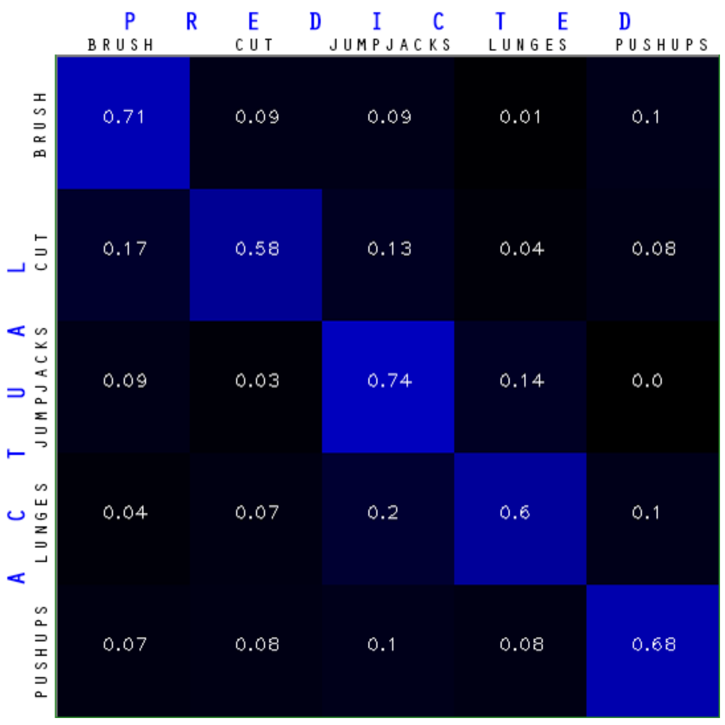
testD1: Learning Rate Tuning	
Learning Rate	**0.0001**
Dropout Rate	**0.5**
Batch Size	**15**
Num of Epochs	10000

Class	**0**	**1**	**2**	**3**	**4**
Precision	0.706	0.575	0.740	0.598	0.679
Recall	0.662	0.679	0.587	0.691	0.708
F-Score	0.684	0.622	0.654	0.641	0.694

Average Precision	0.6596
Standard Dev.	0.0632

Average Recall	0.6654
Standard Dev.	0.0420

Average F-Score	0.6590
Standard Dev.	0.0267



ACTUAL \ PREDICTED	BRUSH	CUT	JUMPJACKS	LUNGES	PUSHUPS
BRUSH	0.71	0.09	0.09	0.01	0.1
CUT	0.17	0.58	0.13	0.04	0.08
JUMPJACKS	0.09	0.03	0.74	0.14	0.0
LUNGES	0.04	0.07	0.2	0.6	0.1
PUSHUPS	0.07	0.08	0.1	0.08	0.68

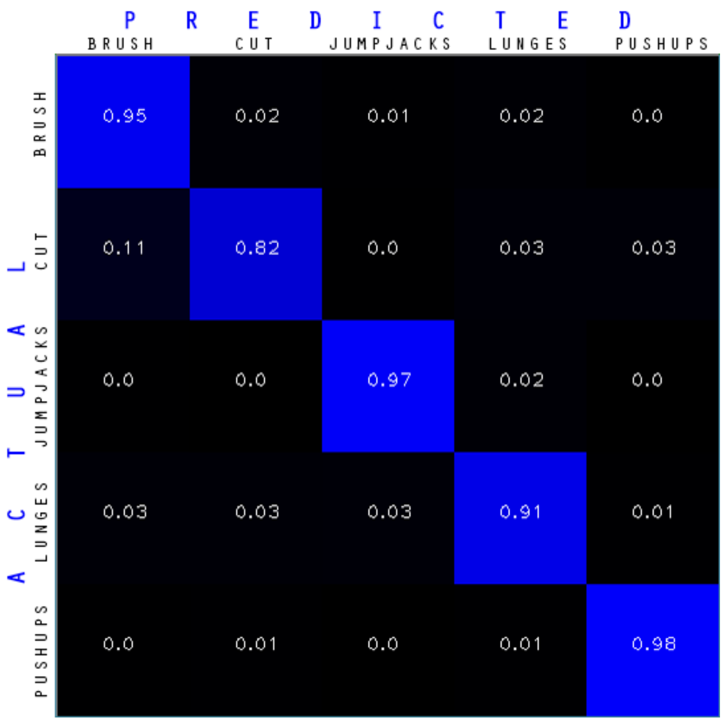
testD2: Learning Rate Tuning	
Learning Rate	**0.002**
Dropout Rate	**0.5**
Batch Size	**15**
Num of Epochs	10000

Class	**0**	**1**	**2**	**3**	**4**
Precision	0.947	0.822	0.971	0.906	0.981
Recall	0.867	0.934	0.957	0.916	0.957
F-Score	0.905	0.874	0.964	0.911	0.969

Average Precision	0.9254
Standard Dev.	0.0578

Average Recall	0.9262
Standard Dev.	0.0333

Average F-Score	0.9246
Standard Dev.	0.0365



ACTUAL \ PREDICTED	BRUSH	CUT	JUMPJACKS	LUNGES	PUSHUPS
BRUSH	0.95	0.02	0.01	0.02	0.0
CUT	0.11	0.82	0.0	0.03	0.03
JUMPJACKS	0.0	0.0	0.97	0.02	0.0
LUNGES	0.03	0.03	0.03	0.91	0.01
PUSHUPS	0.0	0.01	0.0	0.01	0.98

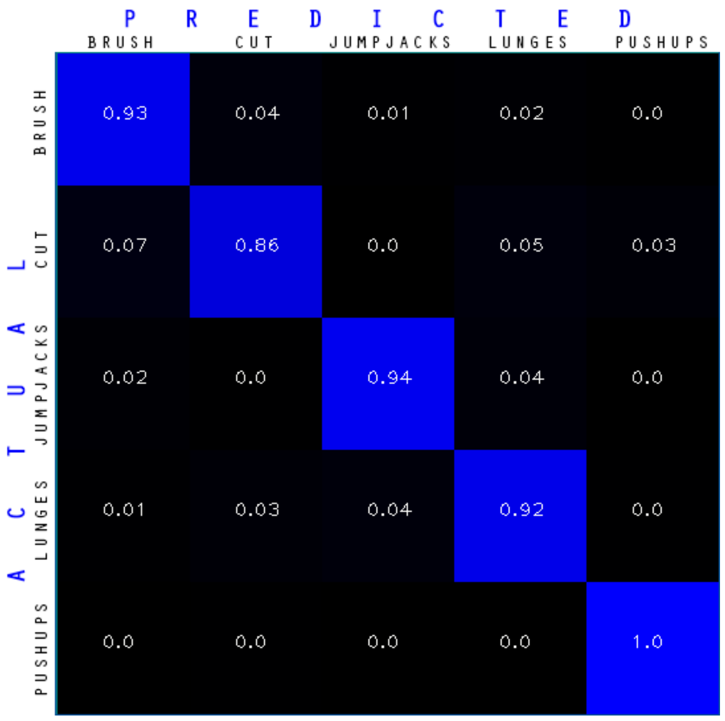
testD3: Learning Rate Tuning	
Learning Rate	**0.005**
Dropout Rate	**0.5**
Batch Size	**15**
Num of Epochs	10000

Class	**0**	**1**	**2**	**3**	**4**
Precision	0.931	0.858	0.939	0.917	0.996
Recall	0.906	0.917	0.948	0.904	0.966
F-Score	0.918	0.887	0.944	0.910	0.981

Average Precision	0.9282
Standard Dev.	0.0442

Average Recall	0.9282
Standard Dev.	0.0246

Average F-Score	0.9280
Standard Dev.	0.0322



ACTUAL \ PREDICTED	BRUSH	CUT	JUMPJACKS	LUNGES	PUSHUPS
BRUSH	0.93	0.04	0.01	0.02	0.0
CUT	0.07	0.86	0.0	0.05	0.03
JUMPJACKS	0.02	0.0	0.94	0.04	0.0
LUNGES	0.01	0.03	0.04	0.92	0.0
PUSHUPS	0.0	0.0	0.0	0.0	1.0

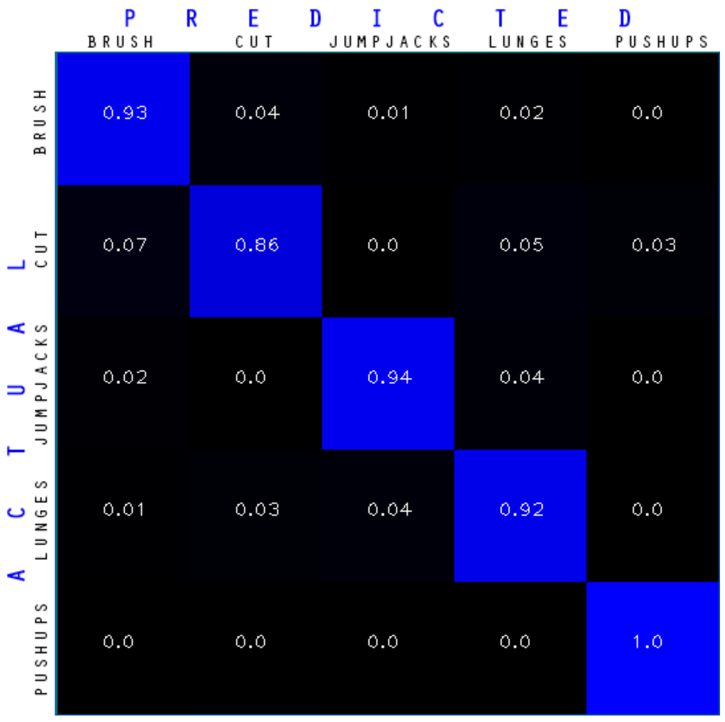
testD4: Learning Rate Tuning	
Learning Rate	**0.05**
Dropout Rate	**0.5**
Batch Size	**15**
Num of Epochs	10000

Class	**0**	**1**	**2**	**3**	**4**
Precision	0.931	0.858	0.939	0.917	0.996
Recall	0.905	0.917	0.948	0.903	0.966
F-Score	0.918	0.887	0.944	0.911	0.981

Average Precision	0.9282
Standard Dev.	0.0442

Average Recall	0.9278
Standard Dev.	0.0250

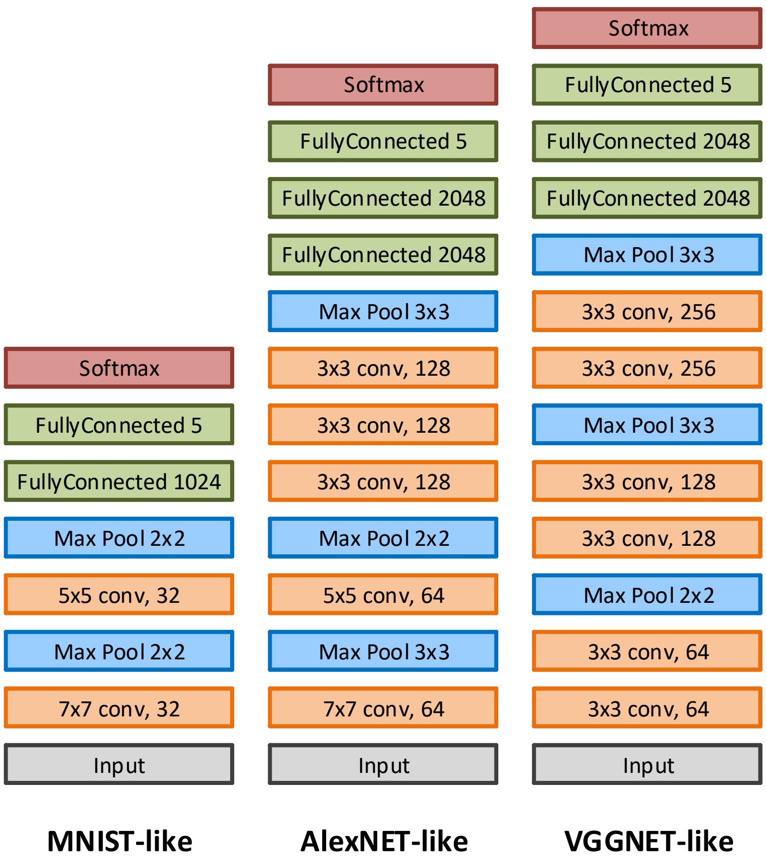
Average F-Score	0.9282
Standard Dev.	0.0320



ACTUAL \ PREDICTED	BRUSH	CUT	JUMPJACKS	LUNGES	PUSHUPS
BRUSH	0.93	0.04	0.01	0.02	0.0
CUT	0.07	0.86	0.0	0.05	0.03
JUMPJACKS	0.02	0.0	0.94	0.04	0.0
LUNGES	0.01	0.03	0.04	0.92	0.0
PUSHUPS	0.0	0.0	0.0	0.0	1.0

Topology Training:

The best parameters from the tuning tests above are kept as the final setup for training on bigger/deeper networks. A more detailed explanation of the choices is given in the Discussion, but ultimately a Batch Size of 15, Dropout Rate of 0.5, and Learning Rate of 0.005 are used for all future tests. The next step is to analyse performance of different network topologies. Three networks were tested:



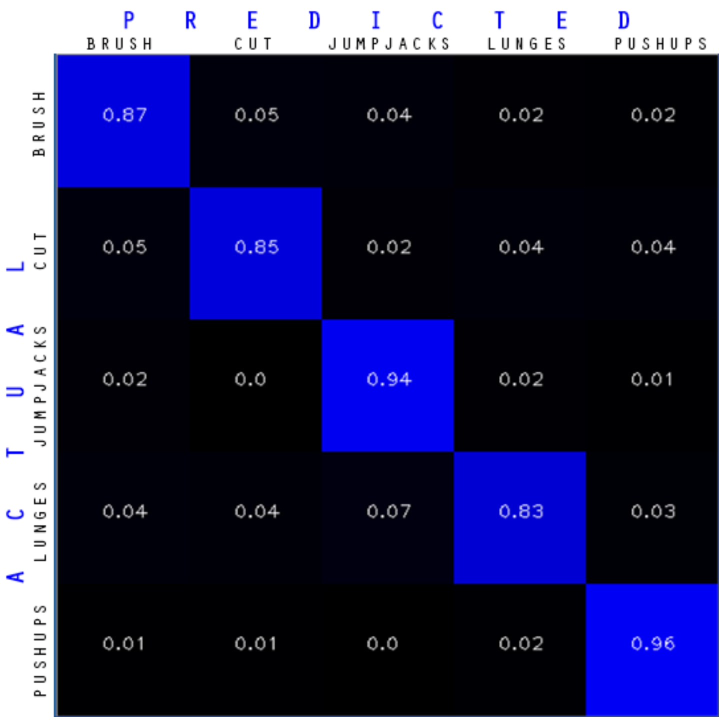
testE1: MNIST-Based Topology	
Learning Rate	0.005
Dropout Rate	0.5
Batch Size	15
Num of Epochs	10000

Class	**0**	**1**	**2**	**3**	**4**
Precision	0.874	0.855	0.943	0.825	0.962
Recall	0.878	0.898	0.874	0.896	0.913
F-Score	0.876	0.876	0.907	0.859	0.937

Average Precision	0.8918
Standard Dev.	0.0523

Average Recall	0.8918
Standard Dev.	0.0142

Average F-Score	0.8910
Standard Dev.	0.0277



ACTUAL \ PREDICTED	BRUSH	CUT	JUMPJACKS	LUNGES	PUSHUPS
BRUSH	0.87	0.05	0.04	0.02	0.02
CUT	0.05	0.85	0.02	0.04	0.04
JUMPJACKS	0.02	0.0	0.94	0.02	0.01
LUNGES	0.04	0.04	0.07	0.83	0.03
PUSHUPS	0.01	0.01	0.0	0.02	0.96

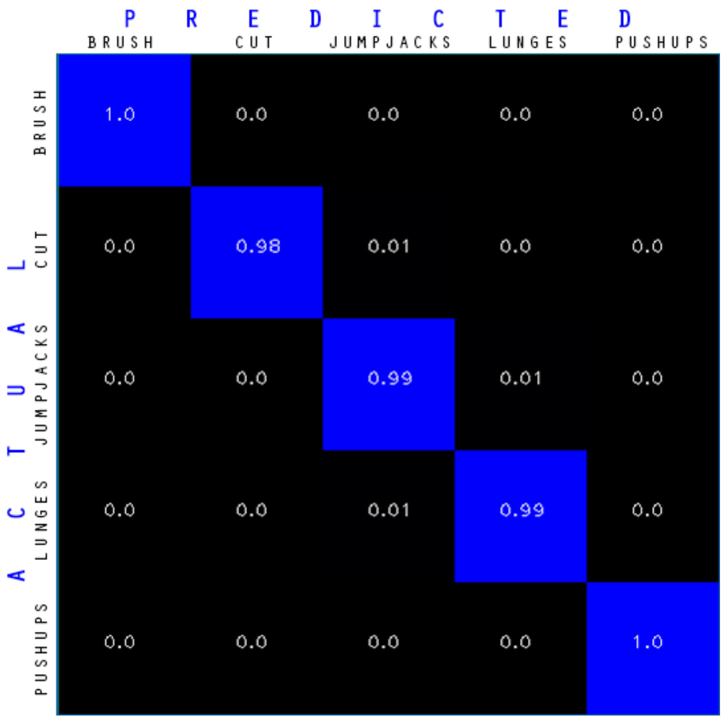
testE2: AlexNET-Based Topology	
Learning Rate	0.005
Dropout Rate	0.5
Batch Size	15
Num of Epochs	10000

Class	**0**	**1**	**2**	**3**	**4**
Precision	0.996	0.982	0.987	0.988	0.996
Recall	0.995	0.996	0.976	0.983	0.999
F-Score	0.995	0.989	0.982	0.986	0.998

Average Precision	0.9898
Standard Dev.	0.0055

Average Recall	0.9898
Standard Dev.	0.0088

Average F-Score	0.9900
Standard Dev.	0.0058



ACTUAL \ PREDICTED	BRUSH	CUT	JUMPJACKS	LUNGES	PUSHUPS
BRUSH	1.0	0.0	0.0	0.0	0.0
CUT	0.0	0.98	0.01	0.0	0.0
JUMPJACKS	0.0	0.0	0.99	0.01	0.0
LUNGES	0.0	0.0	0.01	0.99	0.0
PUSHUPS	0.0	0.0	0.0	0.0	1.0

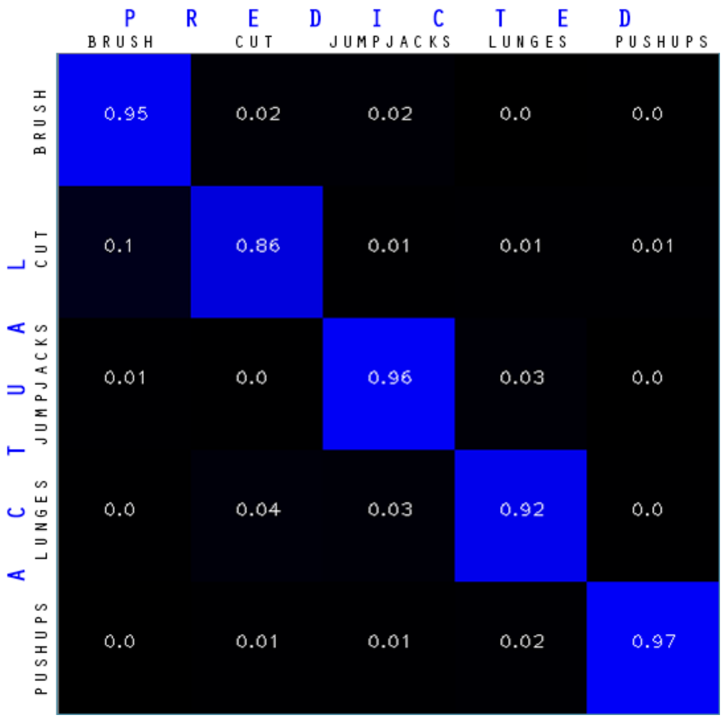
testE3: VGGNET-Based Topology	
Learning Rate	0.005
Dropout Rate	0.5
Batch Size	15
Num of Epochs	10000

Class	**0**	**1**	**2**	**3**	**4**
Precision	0.950	0.863	0.959	0.921	0.969
Recall	0.894	0.921	0.931	0.941	0.978
F-Score	0.922	0.891	0.945	0.931	0.974

Average Precision	0.9324
Standard Dev.	0.0382

Average Recall	0.9330
Standard Dev.	0.0274

Average F-Score	0.9326
Standard Dev.	0.0272

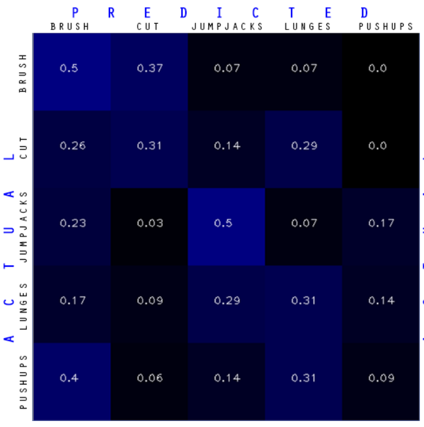


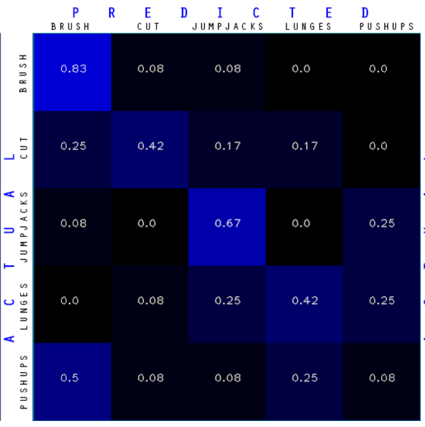
ACTUAL \ PREDICTED	BRUSH	CUT	JUMPJACKS	LUNGES	PUSHUPS
BRUSH	0.95	0.02	0.02	0.0	0.0
CUT	0.1	0.86	0.01	0.01	0.01
JUMPJACKS	0.01	0.0	0.96	0.03	0.0
LUNGES	0.0	0.04	0.03	0.92	0.0
PUSHUPS	0.0	0.01	0.01	0.02	0.97

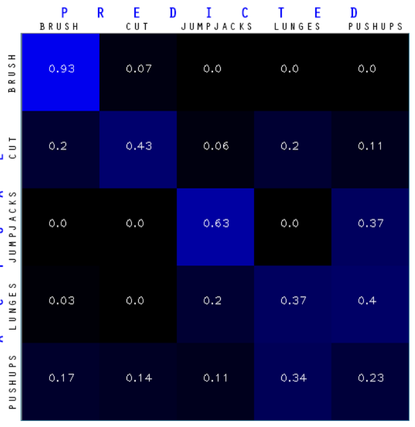
Test Set – Single Frame:

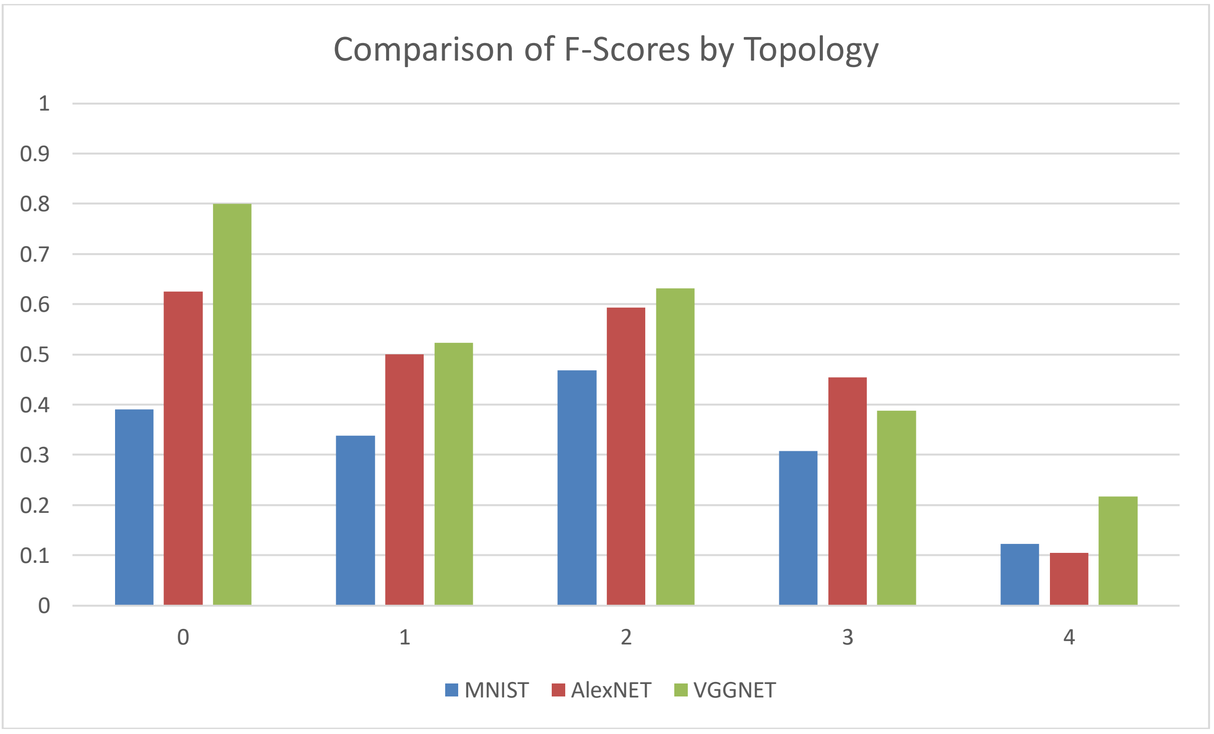
Having determined three functional topologies, and checked their performance on validation sets, the next step is to pass the test set into the generated models and check prediction performance on videos in the test set. Each topology will have created a model per fold during cross validation, so tests will run on each model and generate an average confusion matrix. **The Test Set is the set of personally collected videos.**

Topology	Class	0	1	2	3	4
MNIST-Based	*Precision*	0.5	0.3143	0.5	0.3143	0.0857
	Recall	0.3201	0.3667	0.4393	0.2999	0.2169
	F-Score	0.3903	0.3385	0.4677	0.3070	0.1229
AlexNET-Based	*Precision*	0.8333	0.4167	0.6668	0.4167	0.0833
	Recall	0.5	0.6250	0.5333	0.4999	0.1429
	F-Score	0.6250	0.5	0.5926	0.4545	0.1053
VGGNET-Based	*Precision*	0.9333	0.4286	0.6333	0.3714	0.2286
	Recall	0.7	0.6716	0.6303	0.4063	0.2060
	F-Score	0.7999	0.5233	0.6318	0.3881	0.2167





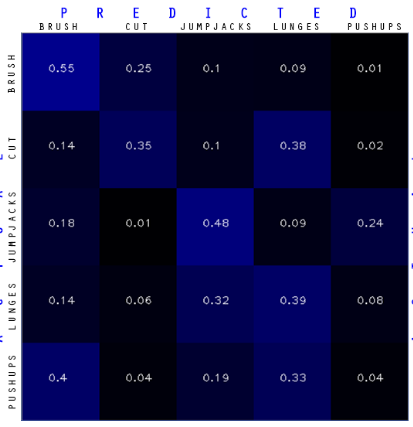




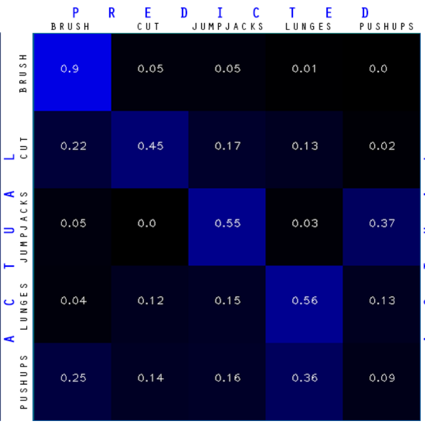
Test Set – All Frames:

Since the Test Set is overall rather limited (low number of videos available per class), to gain a better understanding on the performance of topologies on the given videos, it would be beneficial to run the same kind of test, but on every single frame of the test videos. This gives a more accurate perspective on how the topology would react to any given frame.

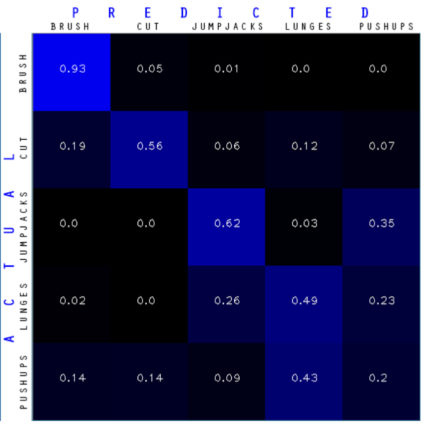
Topology	Class	0	1	2	3	4
MNIST-Based	*Precision*	0.5503	0.3457	0.4840	0.3944	0.0372
	Recall	0.3899	0.4869	0.4044	0.3061	0.0945
	F-Score	0.4564	0.4044	0.4406	0.3447	0.0534
AlexNET-Based	*Precision*	0.8974	0.4490	0.5488	0.5646	0.0925
	Recall	0.6137	0.5889	0.5121	0.5155	0.1515
	F-Score	0.7289	0.5095	0.5299	0.5385	0.1149
VGGNET-Based	*Precision*	0.9338	0.5591	0.6201	0.4862	0.1989
	Recall	0.7252	0.7442	0.5934	0.4546	0.2348
	F-Score	0.8164	0.7385	0.6064	0.4698	0.2153



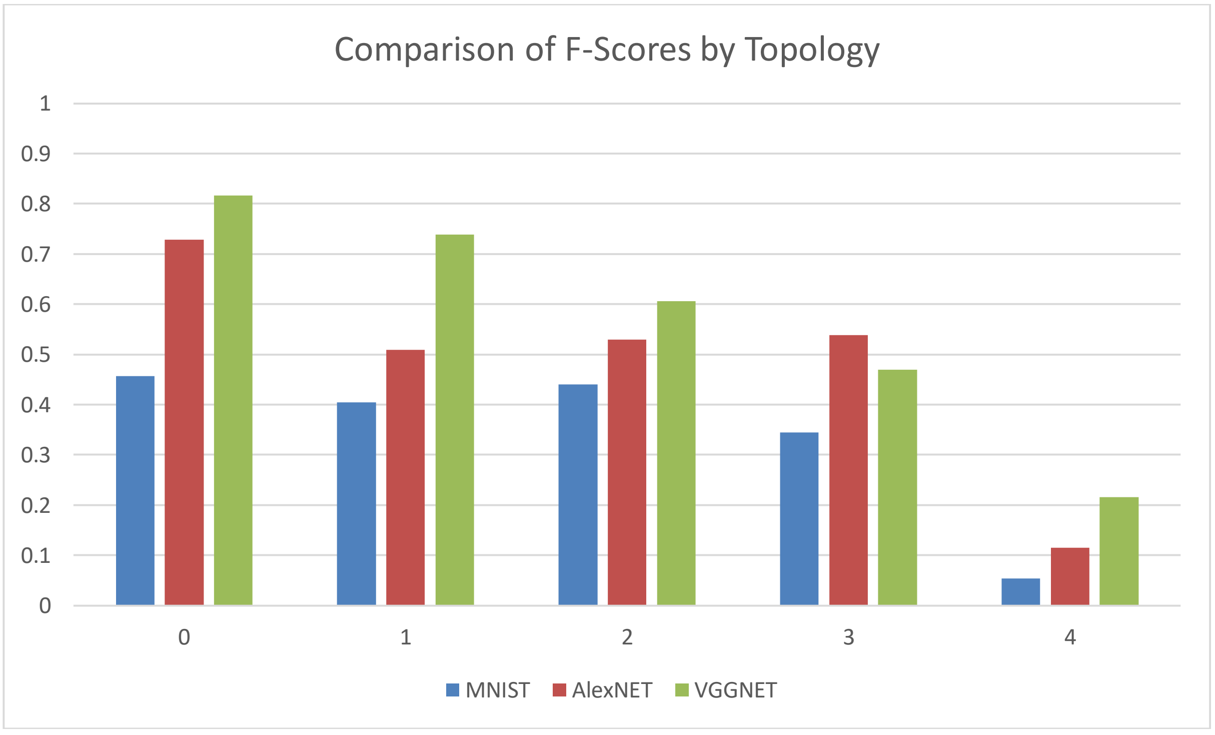
	BRUSH	CUT	JUMPJACKS	LUNGES	PUSHUPS
BRUSH	0.55	0.25	0.1	0.09	0.01
CUT	0.14	0.35	0.1	0.38	0.02
JUMPJACKS	0.18	0.01	0.48	0.09	0.24
LUNGES	0.14	0.06	0.32	0.39	0.08
PUSHUPS	0.4	0.04	0.19	0.33	0.04



	BRUSH	CUT	JUMPJACKS	LUNGES	PUSHUPS
BRUSH	0.9	0.05	0.05	0.01	0.0
CUT	0.22	0.45	0.17	0.13	0.02
JUMPJACKS	0.05	0.0	0.55	0.03	0.37
LUNGES	0.04	0.12	0.15	0.56	0.13
PUSHUPS	0.25	0.14	0.16	0.36	0.09



	BRUSH	CUT	JUMPJACKS	LUNGES	PUSHUPS
BRUSH	0.93	0.05	0.01	0.0	0.0
CUT	0.19	0.56	0.06	0.12	0.07
JUMPJACKS	0.0	0.0	0.62	0.03	0.35
LUNGES	0.02	0.0	0.26	0.49	0.23
PUSHUPS	0.14	0.14	0.09	0.43	0.2



Discussion

Parameter Tuning:

During this section, we took three parameters of our convolutional neural network – Batch Size, Dropout Rate, and Learning Rate – and iteratively ran different tests on those parameters to attempt to tune the network for optimum performance in our domain. At this stage of the process, the test set (which consists of our own videos) was set aside and remained unused for the duration of the validation tests. k-fold cross validation with k = 5 was performed on the training set, using the "leave one out" strategy. While five folds were used for our tests, the method that handles cross validation is of course scalable to any number of folds.

The most immediately noticeable phenomenon in the validation process is the high performance of the classifier on the validation set. The F-score was consistently above 0.9, indicating precise and robust performance on the data used for validation. The confusion matrices reflect this performance, with the largest concentration of classifications being along the diagonal. For some categories, namely the wall push-ups class, the classifier categorised the validation examples correctly at least 98% of the time almost consistently. Needless to say, this is not necessarily desirable, as it could indicate overfitting. Whether or not this happened, and to what extent, will become more apparent when running the framework on the test set.

To begin with, the network was run on both the barebones training data and the augmented training data, where the latter also contained all of the video frames of the former but flipped along the horizontal. This artificial method of adding more training data has been shown to help generalisation of the classifier, but in our case, there was no significant increase in classifier performance during validation. Despite this, it was decided to continue to use the augmented data because, on paper, it should allow the network to better generalise on unseen data.

The Batch Size indicated how many training examples are present in a single batch that is propagated through the network. In our domain, this parameter would correspond to the number of video frames or sets of video frames per iteration. Too large or too small and the network might struggle to generalise[1]. Due to the susceptibility of this parameter to have an adverse effect on the performance of the network on unseen data, we chose it as a parameter for tuning. In our tests, the batch size of 15 outperformed other batch sizes slightly, and that value was therefore used for subsequent parameter tuning tests.

The Dropout Rate is a parameter used specifically to avoid overfitting. It indicates the probability that a neuron in a hidden layer will be "dropped out" of the network (set to a 0 value) during learning. Without this strategy of dropping out hidden neurons, the network's neurons would be liable to becoming co-dependent and creating overly precise and complex patterns to fit the data. Stochastically dropping hidden neurons during learning iterations will force the neurons to be able to generalise more in the potential absence of other neuron weights in the same layer. In practice, it was found during tuning that a Dropout Rate of 0.5 (that is, 50% chance for a hidden neuron to be dropped out) produced the best performance, which is in accordance with previous research[2].

Finally, the Learning Rate parameter simply dictates how much the weights will "move" in the direction of the gradient descent during backpropagation. Too large of a learning rate will cause the network to struggle to converge due to overshooting, but too small of a value and the network will take an infeasibly long time

[1] https://arxiv.org/abs/1609.04836
[2] https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

to converge. We can see the result of this in testD1 of the parameter tuning stage, where a learning rate of 0.0001 was tested. The classifier's performance was at its worst for this test, since it simply had not finished converging yet and would need a larger number of epochs to converge when taking such small steps towards the minimum of the loss function.

Testing:

Upon completion of the parameter tuning, the test set (that has, thus far, been completely unseen) was used to judge the performance of the classifier. Three different convolutional neural network topologies of varying complexities were trained and tested: MNIST-based, AlexNET-based, and VGGNET-based. Overall, the best performance was gleaned from the most complex topology, the VGGNET-based network topology. This is in accordance with the fact that VGGNET is seen as one of the state-of-the-art topologies in the field.

The most noticeable (and perhaps unsurprising) outcome is that the performance on the test set is drastically worse than the performance during validation. It was postulated that the reason for such high classification accuracy during validation compared to the test set likely stems from how the data set used for training was set up. Each video file in the training set is actually a small time slice of a longer video, which means that many features and characteristics of a single video file in the data set (viewpoint, lighting, colours, and so on) are shared across multiple other video files that correspond to the same longer video. When performing cross validation, these short video files will be randomly scattered into different folds, which means that the classifier could be getting trained using video files that are potentially highly similar to the video files it is being validated on. This problem is compounded even further when introducing the augmented data (i.e. the flipped video frames). Of course, this anomaly would not present when we introduce the unseen test data for classification, so the performance of the classifier is bound to plummet to some degree.

A rather exceptional case to make observations on is the performance of the wall push-ups class. Accuracy on classifying wall push-ups went from an almost consistent 98% during validation to practically as good as or even worse than random classification (20%) during testing on all network topologies. This is a clear example of overfitting. The neural network's weights were fitted too precisely to the data found in the training set, and when it came to classifying the unseen data, it was not able to generalise. Finding out why exactly it has such poor performance is unfortunately difficult, due to the black-box nature of the abstraction that neural networks perform in the hidden layers, but no doubt is it an issue that could be solved with more training examples of wall push-ups and more parameter.

Another interesting point of discussion is the confusion matrices. These not only reflect the performance on classifying testing examples accurately, but they also show where the network misclassified the data. The confusion matrix of the best-performing network topology, VGGNet-based, testing on all frames shows impressively high accuracy for brushing teeth (93%) and respectable performance on cutting vegetables (56%) and jumping jacks (62%), but enters a more grey area for lunges and push-ups. In fact, for the most part, the network confused all three classes involving physical exercise with each other more so than with brushing teeth or cutting vegetables. This intuitively makes sense, since the physical exercise videos share a number of characteristics in the data set, such as being more likely to include the full human body or containing larger or more obvious bodily motions than brushing teeth or cutting vegetables.

Overall, the CNN framework developed for this assignment yielded interesting results and provided us with intriguing insights into the benefits of parameter tuning, the effects of overfitting, and the general difficulty and ambiguity involved in video classification.