Mubarik Mohamoud
6.835 Project 2

Gesture recognition Using the Kinect

Question 1: The joint angles, the hand positioning, and most commonly the speeds seem to vary among sequences of the same type. As the poses face towards the camera, you see the shoulders getting wider and arms seem to get longer. Since some poses are fairly similar, varying starting points would have probably confused any classification system. This would cause even more problems for nearest neighbor classifier due to it is heavy reliance on distances.

Question 2: By simply guessing the labels, there is $\frac{1}{9} = 11.11\%$ chance of correctly guessing any label which is in other words the expected accuracy of simply guessing. That's not the best possible accuracy you can get by simply guessing in a given run, but I assumed the questions is looking for the expected value.

Question 3: The $L2$ distance computation involves vector subtractions which you can only perform on vectors of the same length.

Upon implementing normalized frames `normalize_frames(gesture_sets, num_frames)`, there're two situations to handle: 1) **Too long**: When a given frame length is greater then `num_frames` 2) **Too short**: When a given frame length is less than `num_frames`.

Too Long: I calculate `step` and `offset` as follows:
`step =  frames/num_remove;` where `frames= len(sequence.frames);` and `num_remove = frames-num_frames;`

`offset = (step+frames%num_remove)/2;`

I then compute the indices of the frames to be removed as follows.

```
to_remove = [offset+i*step for i in range(num_remove)]
```

It basically says, I want to remove `num_remove` frames spaced by `step` and starting from `offset`. The `step` parameter makes sure that the removed frames are equally spaced where the *`offset`* makes sure that the removed frames are around centered, about same distances from start(index = 0) and the end(index=n-1) of the array.

Too short: Again, compute step and offset as follows:

`step = frames/missing;` where `frames= len(sequence.frames);` and `missing = total-frames;`

`offset = (frames%missing)/2;`

Finally, I compute indices of the frames to be duplicated as follows:

`to_duplicate = [offset+i*step for i in range(missing)];`

The logic behind the `step` and `offset` stays as in the case of **too long**. Note: my current implementation expects `frames>=missing` to hold. This is probably not a good idea and handling the situation where `frames<missing` to should be straightforward, but this seems hold for the given data so I didn't bother.

Question 4: Pros: The nearest neighbor classifier is very simple and fast. Cons: It relies on distances alone for information, which is not the only feature and probably not the most informative feature in this case. Features including relative angles, relative joint speeds, etc could more generally capture what's going in the gesture.

Question 5: The following table shows the results(accuracy) from different pairs of `num_frames` and `ratio` including the best among them(green):

num_frames	ratio	accuracy
20	.3	0.582010582010582
20	.4	0.5987654320987654
20	.6	0.6203703703703703
30	.6	0.8055555555555556
30	.7	0.8271604938271605
40	.7	0.8641975308641975

Question 6: The test accuracy changes because `train_test_split` randomizes the test-train split.

Question 7: The following two for-loops create two dimensional python list of dimensions $30 * 9$ by $33 * num\_frames$ for the features and a $30 * 9$ long array for the labels.

```
for gs in gesture_sets:
    for seq in gs.sequences:
        sample = np.concatenate(list(map(lambda x: x.frame, seq.frames)))
        samples.append(sample)
        labels.append(int(gs.label))
```
The following line converts the label list to numpy array and the list of features to a Numpy matrix.
```
X, Y = np.vstack(samples), np.array(labels)
```

Of course you're doing this because the `sklearn` expects it but that's what's happening.

The following table shows the accuracy different values of `train_size`.

train_size	lowest accuracy	highest accuracy
0.2	45.370370370370374	56.94444444444443
0.3	60.846560846560848	66.137566137566139
0.4	66.049382716049394	69.753086419753089
0.5	62.962962962962962	71.111111111111114
0.6	70.370370370370367	75.925925925925924
0.7	70.370370370370367	75.308641975308646
0.8	74.074074074074076	81.481481481481481
0.9	81.481481481481481	92.592592592592595

Question 8: According to Wikipedia, the Gini impurity measures the frequency of incorrectly labeling a randomly chosen element from a distribution, if it was labeled according to a subset of the distribution. In other words, it has something to do with the likelihood of misclassifying a given element from a distribution of classes or **expected error rate**. It helps the algorithm decide the ordering of the features when building the tree. The following is the formula for the Gini Impurity.

$$I_G(p) = 1 - \sum_{i=1}^{J} p_i^2$$

If p_i is the probability of picking an item class i.

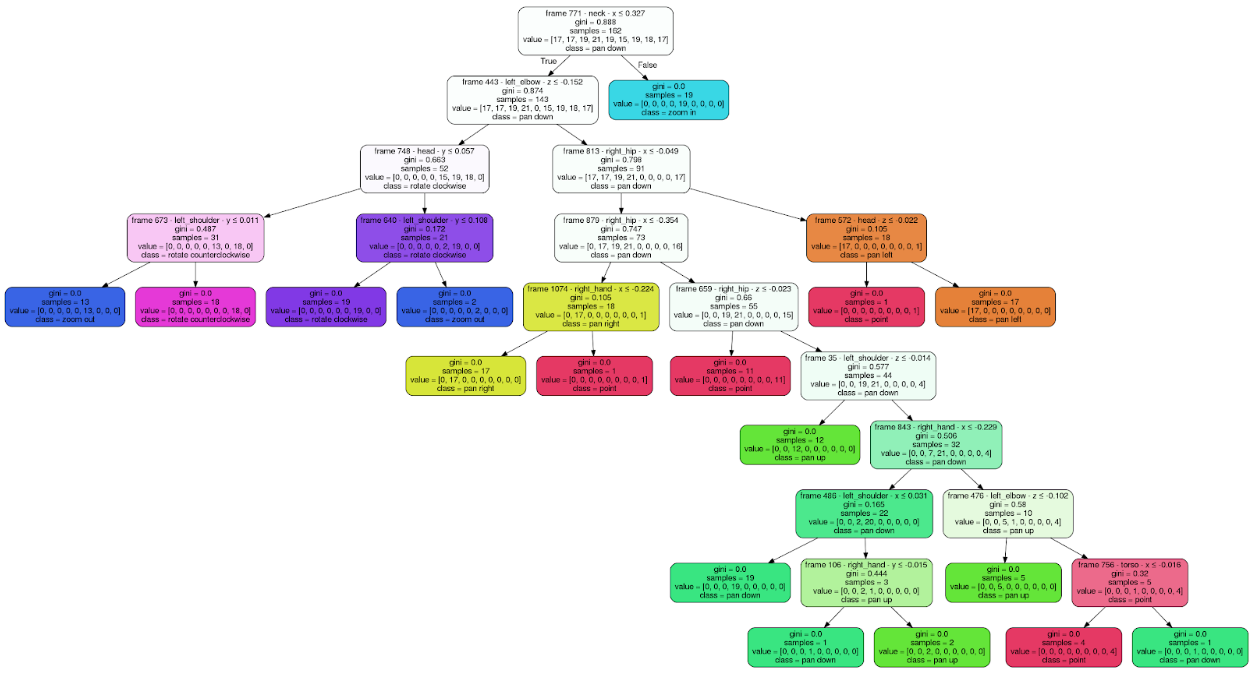
Question 9:

The training accuracy of the tree is **100%** and train accuracy was **70.37%**.
Parameters.

- ***`gini`***: Gini impurity at the node with the given split.

- ***samples***: number of training samples passed to that node(all training samples for the root node).
- ***values***: List of number of samples from each class at the current node.

The impurity is zero at the leaves because each leave classifies the elements of the same class. In other words, the **value** is zero everywhere except one index which makes the probability of misclassifying any element zero.


frame 771 - neck - x ≤ 0.327 gini = 0.888 samples = 162 value = [17, 17, 19, 21, 19, 15, 19, 18, 17] class = pan down
True
False
frame 443 - left_elbow - z ≤ -0.152 gini = 0.874 samples = 143 value = [17, 17, 19, 21, 0, 15, 19, 18, 17] class = pan down
gini = 0.0 samples = 19 value = [0, 0, 0, 0, 19, 0, 0, 0, 0] class = zoom in
frame 748 - head - y ≤ 0.057 gini = 0.663 samples = 52 value = [0, 0, 0, 0, 0, 15, 19, 18, 0] class = rotate clockwise
frame 813 - right_hip - x ≤ -0.049 gini = 0.798 samples = 91 value = [17, 17, 19, 21, 0, 0, 0, 0, 17] class = pan down
frame 673 - left_shoulder - y ≤ 0.011 gini = 0.487 samples = 31 value = [0, 0, 0, 0, 0, 13, 0, 18, 0] class = rotate counterclockwise
frame 640 - left_shoulder - y ≤ 0.108 gini = 0.172 samples = 21 value = [0, 0, 0, 0, 0, 2, 19, 0, 0] class = rotate clockwise
frame 879 - right_hip - x ≤ -0.354 gini = 0.747 samples = 73 value = [0, 17, 19, 21, 0, 0, 0, 0, 16] class = pan down
frame 572 - head - z ≤ -0.022 gini = 0.105 samples = 18 value = [17, 0, 0, 0, 0, 0, 0, 0, 1] class = pan left
gini = 0.0 samples = 13 value = [0, 0, 0, 0, 0, 13, 0, 0, 0] class = zoom out
gini = 0.0 samples = 18 value = [0, 0, 0, 0, 0, 0, 0, 18, 0] class = rotate counterclockwise
gini = 0.0 samples = 19 value = [0, 0, 0, 0, 0, 0, 19, 0, 0] class = rotate clockwise
gini = 0.0 samples = 2 value = [0, 0, 0, 0, 0, 2, 0, 0, 0] class = zoom out
frame 1074 - right_hand - x ≤ -0.224 gini = 0.105 samples = 18 value = [0, 17, 0, 0, 0, 0, 0, 0, 1] class = pan right
frame 659 - right_hip - z ≤ -0.023 gini = 0.66 samples = 55 value = [0, 0, 19, 21, 0, 0, 0, 0, 15] class = pan down
gini = 0.0 samples = 1 value = [0, 0, 0, 0, 0, 0, 0, 0, 1] class = point
gini = 0.0 samples = 17 value = [17, 0, 0, 0, 0, 0, 0, 0, 0] class = pan left
gini = 0.0 samples = 17 value = [0, 17, 0, 0, 0, 0, 0, 0, 0] class = pan right
gini = 0.0 samples = 1 value = [0, 0, 0, 0, 0, 0, 0, 0, 1] class = point
gini = 0.0 samples = 11 value = [0, 0, 0, 0, 0, 0, 0, 0, 11] class = point
frame 35 - left_shoulder - z ≤ -0.014 gini = 0.577 samples = 44 value = [0, 0, 19, 21, 0, 0, 0, 0, 4] class = pan down
gini = 0.0 samples = 12 value = [0, 0, 12, 0, 0, 0, 0, 0, 0] class = pan up
frame 843 - right_hand - x ≤ -0.229 gini = 0.506 samples = 32 value = [0, 0, 7, 21, 0, 0, 0, 0, 4] class = pan down
frame 486 - left_shoulder - x ≤ 0.031 gini = 0.165 samples = 22 value = [0, 0, 2, 20, 0, 0, 0, 0, 0] class = pan down
frame 476 - left_elbow - z ≤ -0.102 gini = 0.58 samples = 10 value = [0, 0, 5, 1, 0, 0, 0, 0, 4] class = pan up
gini = 0.0 samples = 19 value = [0, 0, 0, 19, 0, 0, 0, 0, 0] class = pan down
frame 106 - right_hand - y ≤ -0.015 gini = 0.444 samples = 3 value = [0, 0, 2, 1, 0, 0, 0, 0, 0] class = pan up
gini = 0.0 samples = 5 value = [0, 0, 5, 0, 0, 0, 0, 0, 0] class = pan up
frame 756 - torso - x ≤ -0.016 gini = 0.32 samples = 5 value = [0, 0, 0, 1, 0, 0, 0, 0, 4] class = point
gini = 0.0 samples = 1 value = [0, 0, 0, 1, 0, 0, 0, 0, 0] class = pan down
gini = 0.0 samples = 2 value = [0, 0, 2, 0, 0, 0, 0, 0, 0] class = pan up
gini = 0.0 samples = 4 value = [0, 0, 0, 0, 0, 0, 0, 0, 4] class = point
gini = 0.0 samples = 1 value = [0, 0, 0, 1, 0, 0, 0, 0, 0] class = pan down

Question Bonus 1:

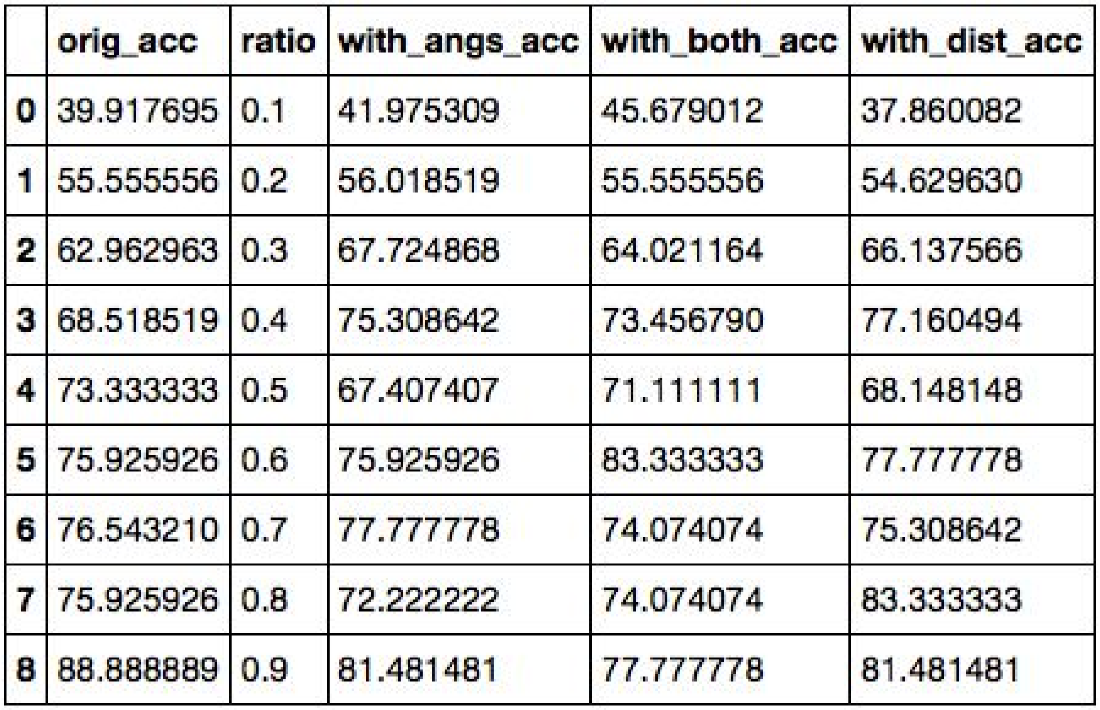
Approach 1: Added scaled-distance ratio between some joint positions as follows.

1. Added the ratio of the distances between the hand and shoulder on the same side to the corresponding arm length. $ratio = \frac{dist(shoulder\ pose,\ hand\ pose)}{arm\ length}$
2. Added the ratio of the distance between the hip and hand on the same side to the corresponding arm length. $ratio = \frac{dist(hip\ pose,\ hand\ pose)}{arm\ length}$.
3. This was done for both right and left side of the body.

Approach 2: Added the angles of the triangles created by the arms where the vertices are the shoulder, elbow, and hand. This was done for both sides of the body.

Approach 3: Add both.

The following table shows the results for each approach vs the original tree. As the table shows, the results are bit better for lower training ratio, but comparable for higher training ratios. This is because the whole idea of adding more features is counter productive. The problem is that the training accuracy is already 100% which means adding more features doesn't change match, in fact it can help overfit even more. I have realized this pretty late so I didn't try to implement an approach that would limit the overfitting problem, but I have done what the staff have suggested so hope it was worth the effort in terms of grade.



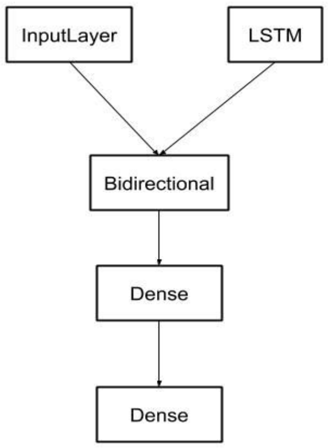
	orig_acc	ratio	with_angs_acc	with_both_acc	with_dist_acc
0	39.917695	0.1	41.975309	45.679012	37.860082
1	55.555556	0.2	56.018519	55.555556	54.629630
2	62.962963	0.3	67.724868	64.021164	66.137566
3	68.518519	0.4	75.308642	73.456790	77.160494
4	73.333333	0.5	67.407407	71.111111	68.148148
5	75.925926	0.6	75.925926	83.333333	77.777778
6	76.543210	0.7	77.777778	74.074074	75.308642
7	75.925926	0.8	72.222222	74.074074	83.333333
8	88.888889	0.9	81.481481	77.777778	81.481481

Question 10: The input hasn't been changed from the Gesture_set representation. The features are arranged in 3D numpy array such that input layer of the network expects two dimensional numpy array(36 frames of 33 features each). This doesn't show that the LSTM is imposing strict requirements compare to the two other algorithms, it shows that the LSTM model architecture is more flexible or more generalized. An interesting observation is that the we don't actually split the data into test and validation for the LSTM beforehand, we only tell it what ratio to use with the validation_split parameter.

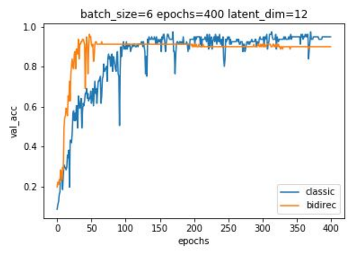
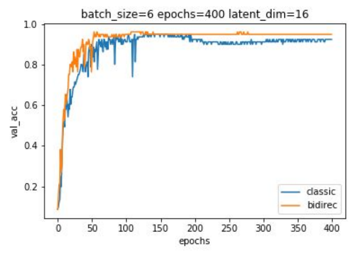
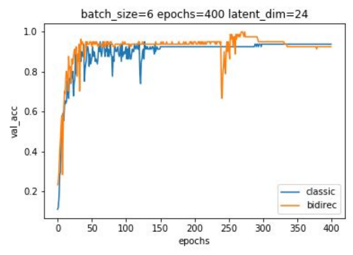
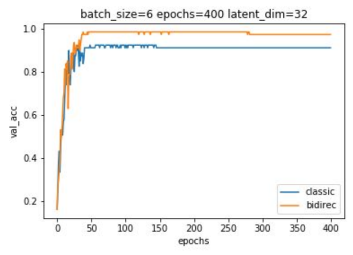
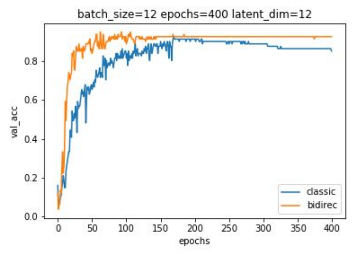
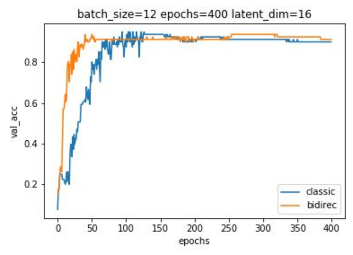
The best accuracy recorded was 95.06% which was found with multiple different parameter sets, but one of them was batch_size=8, epochs=200, and latent_dim=12.

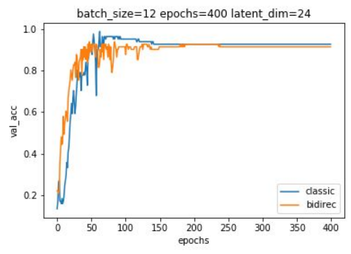
Question 11: I didn't use the hint, I used the **Bidirectional** object as shown by the line of code bilow.

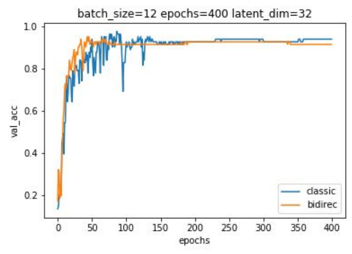
```
lstm = Bidirectional(LSTM(latent_dim))(input_layer)
```

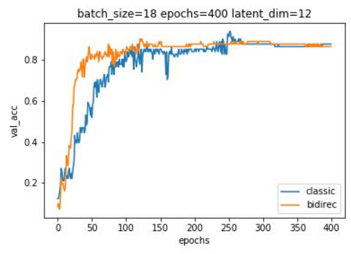


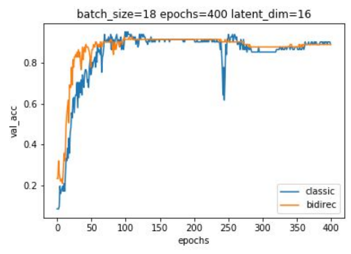
The following table of images show the validation accuracies of the normal LSTM and the Bidirectional LSTM for different values of the batch sizes and hidden layer dimensions as the epochs go from 0-400. Note the axis are label accordingly and the plot labels indicate which plot is which. The plots indicate that the two setups are very much comparable and their relative performances depend on the parameters(batch_size and latent_dim).

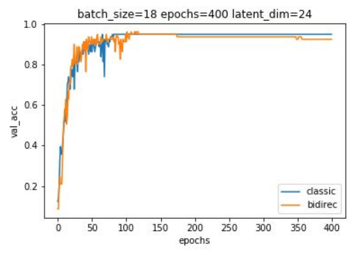
		
		

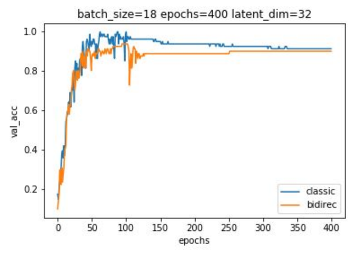

batch_size=12 epochs=400 latent_dim=24
val_acc
epochs
classic
bidirec

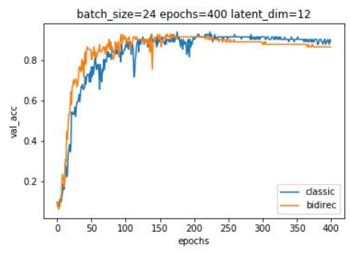

batch_size=12 epochs=400 latent_dim=32
val_acc
epochs
classic
bidirec

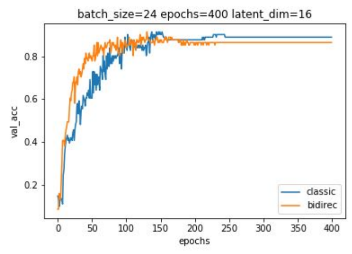

batch_size=18 epochs=400 latent_dim=12
val_acc
epochs
classic
bidirec

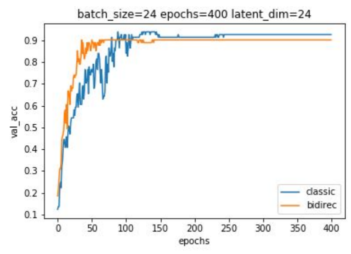

batch_size=18 epochs=400 latent_dim=16
val_acc
epochs
classic
bidirec

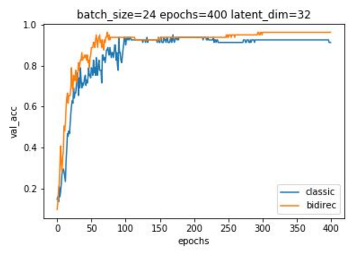

batch_size=18 epochs=400 latent_dim=24
val_acc
epochs
classic
bidirec

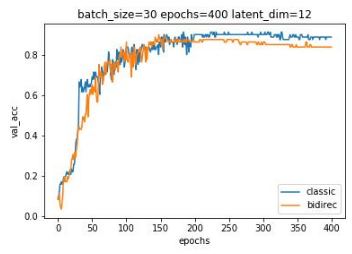

batch_size=18 epochs=400 latent_dim=32
val_acc
epochs
classic
bidirec

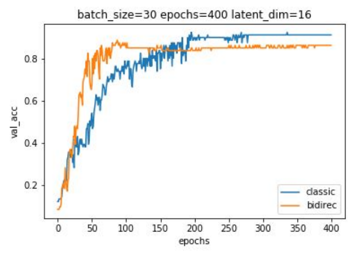

batch_size=24 epochs=400 latent_dim=12
val_acc
epochs
classic
bidirec

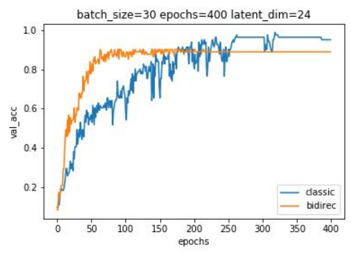

batch_size=24 epochs=400 latent_dim=16
val_acc
epochs
classic
bidirec

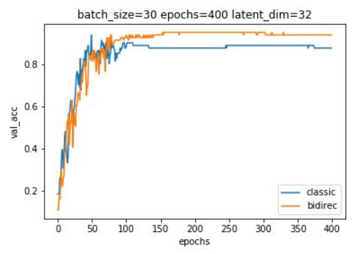

batch_size=24 epochs=400 latent_dim=24
val_acc
epochs
classic
bidirec


batch_size=24 epochs=400 latent_dim=32
val_acc
epochs
classic
bidirec

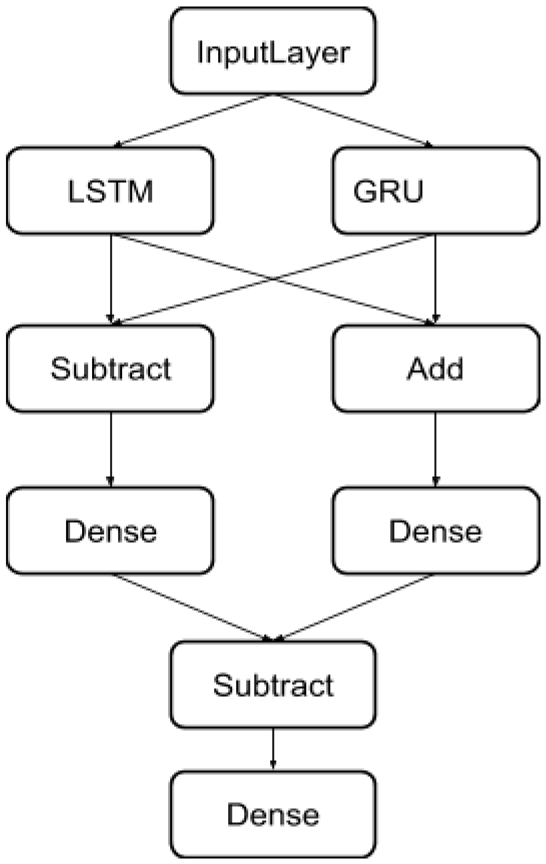

batch_size=30 epochs=400 latent_dim=12
val_acc
epochs
classic
bidirec


batch_size=30 epochs=400 latent_dim=16
val_acc
epochs
classic
bidirec


batch_size=30 epochs=400 latent_dim=24
val_acc
epochs
classic
bidirec


batch_size=30 epochs=400 latent_dim=32
val_acc
epochs
classic
bidirec

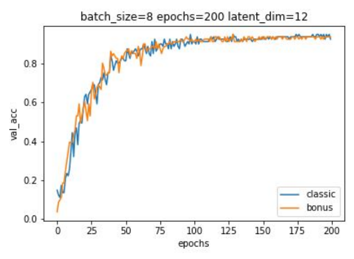
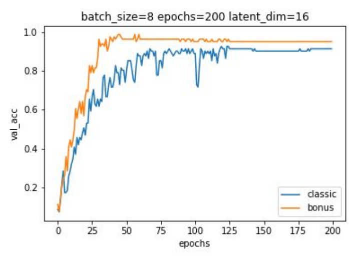
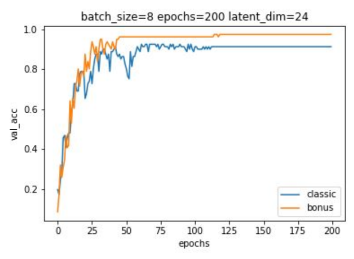
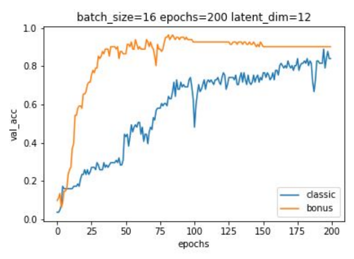
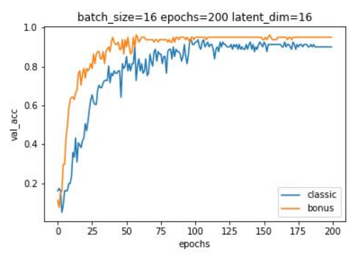
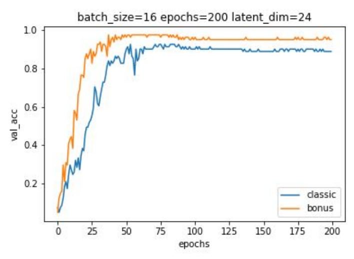
Question bonus 2:

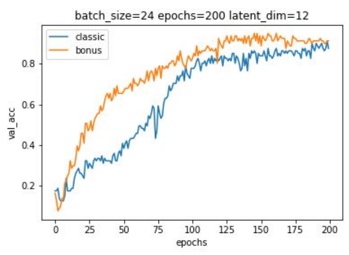


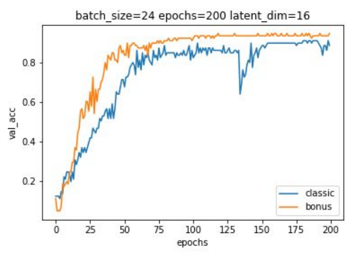
The graph to the left shows the final architecture that produced the best results for the recurrent neural network bonus. The following are the module descriptions.

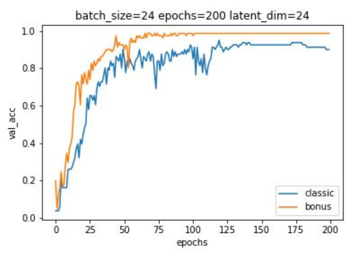
1. **InputLayer**: Inputlayer
2. **LSTM**: LSTM layer
3. **GRU**: Gated Recurrent Unit layer
4. **Subtract**: Merge by Subtraction layer.
5. **Add**: Merge by addition layer
6. **Dense**: Dense layer

The following plots show the performance of the selected bonus architecture against the initial(staff) set. Note: This architecture produce 100% final validation accuracies in some situations.


batch_size=24 epochs=200 latent_dim=12
classic
bonus
val_acc
epochs


batch_size=24 epochs=200 latent_dim=16
classic
bonus
val_acc
epochs


batch_size=24 epochs=200 latent_dim=24
classic
bonus
val_acc
epochs