CC484: Pattern Recognition Assignment #4
Modulation Classification

Faculty of Engineering
Alexandria University
Specialized Scientific Programs
Computer and Communication Engineering

Heba Elwazzan `ID.6521`
Raghad Abo El Eneen `ID.6360`
Omar Aboushousha `ID.6492`

Contents

Abstract

The purpose of this experiment is to explore different neural network architectures on the DeepSig Dataset: RadioML 2016.04C dataset, which is a synthetic dataset, generated with GNU Radio, consisting of 11 modulations. The classes consist of BPSK, QPSK, 8PSK, 16QAM, 64QAM, BFSK, CPFSK, and PAM4 for digital modulations, and WB-FM, and AM-DSB for analog modulations. AM-SSB data seems to be missing. The experiment makes use of different feature spaces including the raw data, first derivative in time, integral in time, and combination of these features. 4 models were built: a CNN architecture with 2 convolutional layers with ReLu activation functions, dense fully connected layer using ReLu, and dense fully SoftMax classifier. Three more models are implemented, a vanilla RNN model, a vanilla LSTM model, and a CLDNN model. The models were then compared in terms of accuracy and the best model is reported. This experiment was implemented in this Google Colab Notebook.

1 Data format

Data is stored in a .dat file in pickled format. When unpickled, a dictionary is revealed, with keys indicating the modulation type and the SNR value of the signal. The signal is composed of a 2x128 array. Plots of some signals are shown in figure 1.

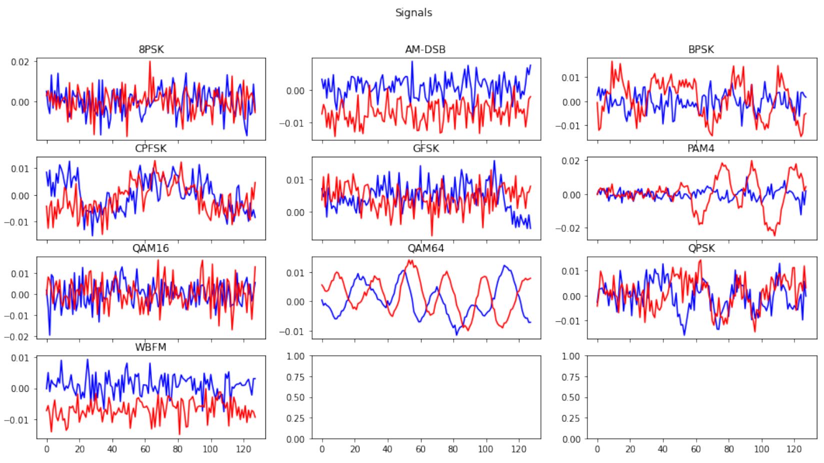


Figure 1: Signal plots in time

2 Data preprocessing techniques

2.1 Label Encoding

Instead of mapping the labels to numerical values, one hot encoding was used to map each category to a binary number, where a 0 indicates "cold" and a "1" indicates hot, removing any correlation between the values in terms of how close the numerical values are to each other.

2.2 Data splitting

Data was split in 70% train\validation dataset, and 30% test dataset. The train dataset was then split into 95% train data, and 5% validation data.

2.3 Data transformation pipeline

Not much in terms of transformation is done. A custom transformer class was created with the purpose of creating the desired feature space based on input. Feature combinations are as follows:

- Raw time series (2 channels).
- First Derivative in time (2 channels)
- Integral in time (2 channels)
- Raw time series + first derivative in time (4 channels)
- Raw time series + integral in time (4 channels)
- first derivative in time + integral in time (4 channels)
- Raw time series + first derivative in time + integral in time (6 channels)

After that, prepared feature space is passed in a pipelined manner to a standard scaler. The output is ready to have a model trained on.

3 Method explanation

7 models are implemented: 3 VT-CNN2 Neural Net models based on [2]; a vanilla RNN model; 2 LSTM models; and a CLDNN model, composed of convolution layers followed by an LSTM, based on [1].

3.1 CNN models

3.1.1 CNN model 1

The figure 2 shows the implemented CNN model: a 2D 64x1x3 convolutional layer with ReLu as an activation function, followed by a 16x2x3 2D convolutional layer with ReLu, followed by a Dense ReLu layer then a Dense Softmax Layer. The convolutional layers used Glorot Uniform as a kernel initializer while the dense layers used HeNormal. Adam optimizer was used to optimize the learning rate and dropout layers were added.

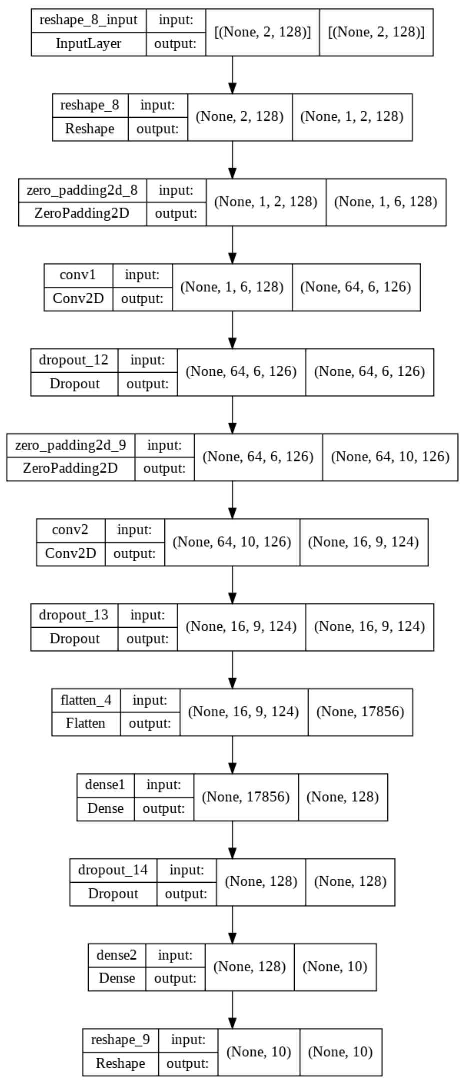

reshape_8_input | InputLayer | input: [(None, 2, 128)] | output: [(None, 2, 128)]
reshape_8 | Reshape | input: (None, 2, 128) | output: (None, 1, 2, 128)
zero_padding2d_8 | ZeroPadding2D | input: (None, 1, 2, 128) | output: (None, 1, 6, 128)
conv1 | Conv2D | input: (None, 1, 6, 128) | output: (None, 64, 6, 126)
dropout_12 | Dropout | input: (None, 64, 6, 126) | output: (None, 64, 6, 126)
zero_padding2d_9 | ZeroPadding2D | input: (None, 64, 6, 126) | output: (None, 64, 10, 126)
conv2 | Conv2D | input: (None, 64, 10, 126) | output: (None, 16, 9, 124)
dropout_13 | Dropout | input: (None, 16, 9, 124) | output: (None, 16, 9, 124)
flatten_4 | Flatten | input: (None, 16, 9, 124) | output: (None, 17856)
dense1 | Dense | input: (None, 17856) | output: (None, 128)
dropout_14 | Dropout | input: (None, 128) | output: (None, 128)
dense2 | Dense | input: (None, 128) | output: (None, 10)
reshape_9 | Reshape | input: (None, 10) | output: (None, 10)

Figure 2: CNN Architecture 1

```
def create_CNN_model(in_shape, dr=0.5, learning_rate=0.001):
    model = models.Sequential()
    model.add(Reshape([1, in_shape[0], in_shape[1]],
        input_shape=in_shape))
    model.add(ZeroPadding2D((0, 2)))
    model.add(Conv2D(64, (1, 3), activation="relu",
        name="conv1", padding="valid",data_format =
        'channels_first',
        kernel_initializer="glorot_uniform"))
    model.add(Dropout(dr))
    model.add(ZeroPadding2D((0, 2)))
    model.add(Conv2D(16, (2, 3), activation="relu",
        name="conv2", padding="valid",data_format =
        'channels_first',
        kernel_initializer="glorot_uniform"))
    model.add(Dropout(dr))
    model.add(Flatten())
    model.add(Dense(128, activation="relu", name="dense1",
        kernel_initializer="he_normal"))
    model.add(Dropout(dr))
    model.add(Dense(10,activation='softmax', name="dense2",
        kernel_initializer="he_normal"))
    optimizer = Adam(learning_rate=learning_rate)
    model.add(Reshape([10]))
    model.compile(loss='categorical_crossentropy',
        optimizer=optimizer, metrics=['accuracy'])
    print(model.summary())
    tf.keras.utils.plot_model(model, "CNN model 1.png",
        show_shapes=True)
    return model
```

3.1.2 CNN model 2

Model 2, shown in figure 3, is similar to model 1, but with different kernel initializers and with batch normalization added.

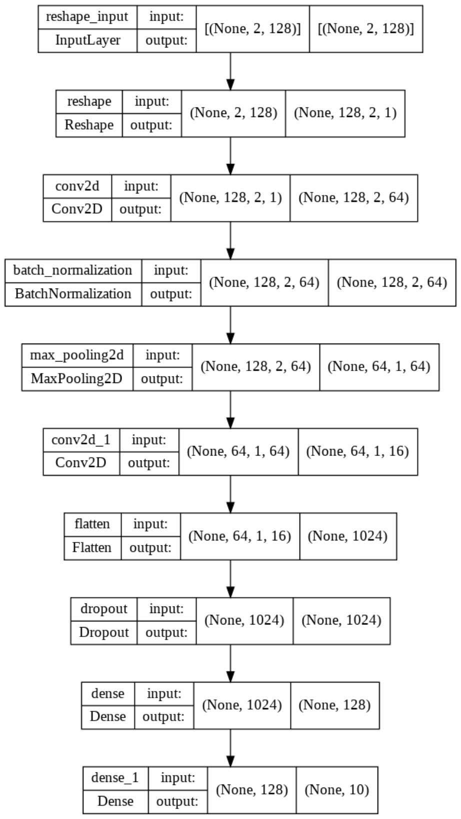

reshape_input | InputLayer | input: [(None, 2, 128)] | output: [(None, 2, 128)]
reshape | Reshape | input: (None, 2, 128) | output: (None, 128, 2, 1)
conv2d | Conv2D | input: (None, 128, 2, 1) | output: (None, 128, 2, 64)
batch_normalization | BatchNormalization | input: (None, 128, 2, 64) | output: (None, 128, 2, 64)
max_pooling2d | MaxPooling2D | input: (None, 128, 2, 64) | output: (None, 64, 1, 64)
conv2d_1 | Conv2D | input: (None, 64, 1, 64) | output: (None, 64, 1, 16)
flatten | Flatten | input: (None, 64, 1, 16) | output: (None, 1024)
dropout | Dropout | input: (None, 1024) | output: (None, 1024)
dense | Dense | input: (None, 1024) | output: (None, 128)
dense_1 | Dense | input: (None, 128) | output: (None, 10)

Figure 3: CNN Architecture 2

```
def create_CNN_model2(in_shape, dr=0.4, learning_rate=0.001):
    conv1_kernel_shape=(3,1)
    conv1_number_of_filters=64
    conv2_kernel_shape=(3,2)
    conv2_number_of_filters=16
    dense1_size = 128
    dense2_size = 10
    dropout = dr

    # Build model
    model_conv = Sequential()
    model_conv.add(Reshape((128,in_shape[0],1),
        input_shape=in_shape))
    model_conv.add(Conv2D(conv1_number_of_filters,
        conv1_kernel_shape, strides=1,
                    padding='same',
                        data_format='channels_last',
                        activation='relu',
                        kernel_initializer='glorot_uniform'))
    model_conv.add(BatchNormalization())
    model_conv.add(MaxPooling2D())
    model_conv.add(Conv2D(conv2_number_of_filters,
        conv2_kernel_shape, strides=1,
                    padding='same',
                        data_format='channels_last',
                        activation='relu',
                        kernel_initializer='glorot_uniform'))
    model_conv.add(Flatten())
    model_conv.add(Dropout(rate=1-dropout))
    model_conv.add(Dense(dense1_size, activation='relu',
        kernel_initializer='glorot_uniform'))
    model_conv.add(Dense(dense2_size, activation='softmax'))

    # Compile model
    model_conv.compile(loss='categorical_crossentropy',
        optimizer='adam', metrics=['accuracy'])
    model_conv.summary()
    Image(tf.keras.utils.plot_model(model_conv, "CNN model
        2.png", show_shapes=True))
    return model_conv
```

3.1.3 CNN model 3

While model 3 may have the same sequence of layers, the layers themselves have more filters. It is directly adapted from [2]'s accompanying notebook. Its structure is show in figure 4.

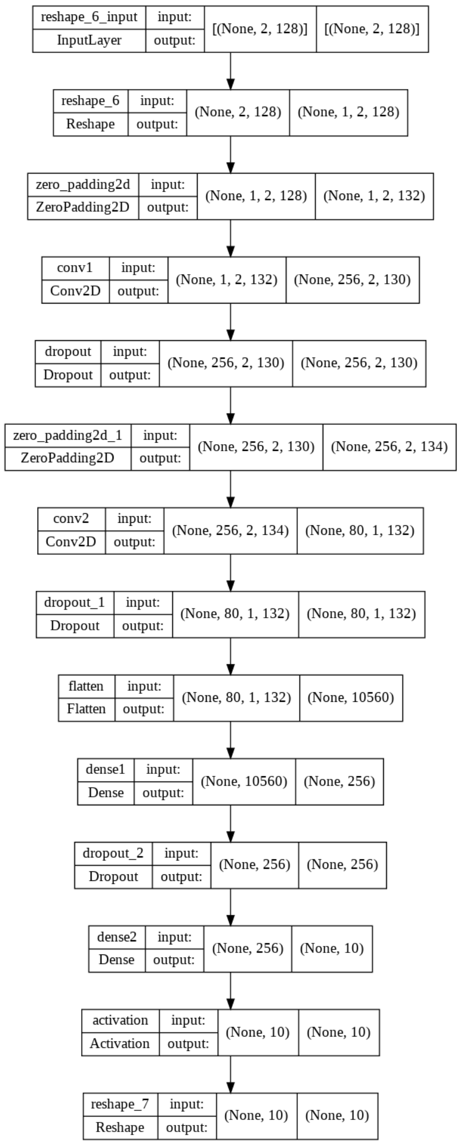

reshape_6_input | InputLayer | input: [(None, 2, 128)] | output: [(None, 2, 128)]
reshape_6 | Reshape | input: (None, 2, 128) | output: (None, 1, 2, 128)
zero_padding2d | ZeroPadding2D | input: (None, 1, 2, 128) | output: (None, 1, 2, 132)
conv1 | Conv2D | input: (None, 1, 2, 132) | output: (None, 256, 2, 130)
dropout | Dropout | input: (None, 256, 2, 130) | output: (None, 256, 2, 130)
zero_padding2d_1 | ZeroPadding2D | input: (None, 256, 2, 130) | output: (None, 256, 2, 134)
conv2 | Conv2D | input: (None, 256, 2, 134) | output: (None, 80, 1, 132)
dropout_1 | Dropout | input: (None, 80, 1, 132) | output: (None, 80, 1, 132)
flatten | Flatten | input: (None, 80, 1, 132) | output: (None, 10560)
dense1 | Dense | input: (None, 10560) | output: (None, 256)
dropout_2 | Dropout | input: (None, 256) | output: (None, 256)
dense2 | Dense | input: (None, 256) | output: (None, 10)
activation | Activation | input: (None, 10) | output: (None, 10)
reshape_7 | Reshape | input: (None, 10) | output: (None, 10)

Figure 4: CNN Architecture 3

```python
def create_CNN_model3(in_shape, dr=0.5, learning_rate=0.001):
    model = models.Sequential()
    model.add(Reshape([1]+in_shape, input_shape=in_shape))
    model.add(ZeroPadding2D(((0, 0), (2, 2)), data_format =
        'channels_first'))
    model.add(Conv2D(256, (1, 3), padding='valid',
        activation="relu", name="conv1", data_format =
        'channels_first',
        kernel_initializer='glorot_uniform'))
    model.add(Dropout(dr))
    model.add(ZeroPadding2D(((0, 0), (2, 2)),data_format =
        'channels_first'))
    model.add(Conv2D(80, (2, 3), padding="valid",
        activation="relu", name="conv2", data_format =
        'channels_first',
        kernel_initializer='glorot_uniform'))
    model.add(Dropout(dr))
    model.add(Flatten())
    model.add(Dense(256, activation='relu',
        kernel_initializer='he_normal', name="dense1"))
    model.add(Dropout(dr))
    model.add(Dense( 10, kernel_initializer='he_normal',
        name="dense2" ))
    model.add(Activation('softmax'))
    model.add(Reshape([10]))
    model.compile(loss='categorical_crossentropy',
        optimizer='adam', metrics=['accuracy'])
    model.summary()
    tf.keras.utils.plot_model(model, "CNN model 3.png",
        show_shapes=True)
    return model
```

3.2 Vanilla RNN model

A simple RNN model was implemented, shown in figure 5. It is composed of two RNN layers of 64 and 128 filters respectively, followed by a dense layer with softmax as an activator.

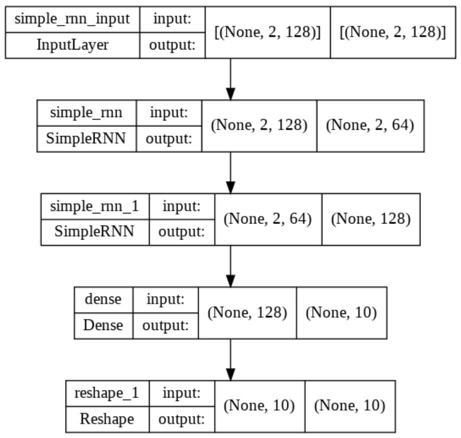


Figure 5: RNN Architecture

```
def create_vanilla_RNN(in_shape, dr=0, learning_rate=0.001):
    model = keras.models.Sequential([
                    keras.layers.SimpleRNN(64,
                        return_sequences=True, dropout=dr,
                        input_shape=in_shape),
                    keras.layers.SimpleRNN(128),
                    keras.layers.Dense(10,activation='softmax',
                        name="dense",
                        kernel_initializer="he_normal"),
                    Reshape([10])
    ])
    optimizer = Adam(learning_rate=learning_rate)
    model.compile(loss='categorical_crossentropy',
        optimizer=optimizer, metrics=['accuracy'])
    print(model.summary())
    tf.keras.utils.plot_model(model, "RNN model 1.png",
        show_shapes=True)
    return model
```

3.3 LSTM model

3.3.1 LSTM model 1: 2 LSTM layers

A simple LSTM, shown in figure 6, composed of 2 LSTM layers and a dense layer.

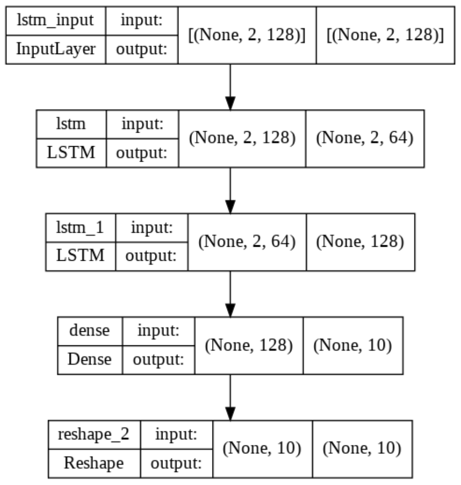
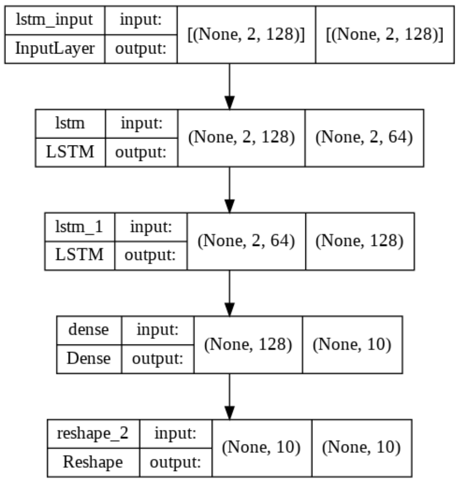


Figure 6: LSTM Architecture 1

```
def create_LSTM(in_shape, dr=0, learning_rate=0.001):
    model = keras.models.Sequential([
                keras.layers.LSTM(64,
                    return_sequences=True,
                    input_shape=in_shape),
                keras.layers.LSTM(128, dropout=dr),
                keras.layers.Dense(10,activation='softmax',
                    name="dense",
                    kernel_initializer="he_normal"),
                Reshape([10])
    ])
    optimizer = Adam(learning_rate=learning_rate)
    model.compile(loss='categorical_crossentropy',
        optimizer=optimizer, metrics=["accuracy"])
    print(model.summary())
    tf.keras.utils.plot_model(model, "LSTM model 1.png",
        show_shapes=True)
    return model
```

3.3.2 LSTM model 2: 3 LSTM layers

The first LSTM model was modified by adding an extra LSTM layer to see if performance is improved by making the model deeper, and shown in figure 7.



lstm_input | input: | output: | InputLayer | [(None, 2, 128)] | [(None, 2, 128)]
lstm | LSTM | input: | output: | (None, 2, 128) | (None, 2, 64)
lstm_1 | LSTM | input: | output: | (None, 2, 64) | (None, 128)
dense | Dense | input: | output: | (None, 128) | (None, 10)
reshape_2 | Reshape | input: | output: | (None, 10) | (None, 10)

Figure 7: LSTM Architecture 2

```
def create_LSTM2(in_shape, dr=0, learning_rate=0.001):
    model = keras.models.Sequential([
                    keras.layers.LSTM(64,
                        return_sequences=True,
                        input_shape=in_shape),
                    keras.layers.LSTM(128, dropout=dr,
                        return_sequences=True),
                    keras.layers.LSTM(256, dropout=dr),
                    keras.layers.Dense(10,activation='softmax',
                        name="dense",
                        kernel_initializer="glorot_uniform"),
                    Reshape([10])
    ])
    optimizer = Adam(learning_rate=learning_rate)
    model.compile(loss='categorical_crossentropy',
        optimizer=optimizer, metrics=["accuracy"])
    print(model.summary())
    tf.keras.utils.plot_model(model, "LSTM model 2.png",
        show_shapes=True)
    return model
```

3.4 CLDNN model

The architecture shown in figure 8 was adapted directly from [1]'s take on the CLDNN model. 3 1D convolutional layers were followed by LSTM layers then one dense fully connected layer. Dropout was added to avoid overfitting the data. Kernel initializers were used (HeNormal for the dense layer and Glorot Uniform otherwise).

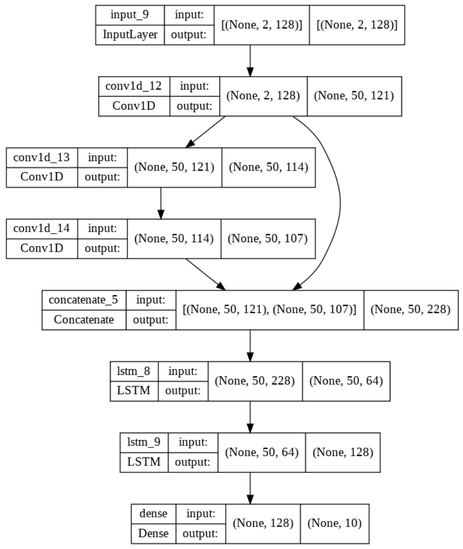

input_9 | InputLayer | input: | output: | [(None, 2, 128)] | [(None, 2, 128)]
conv1d_12 | Conv1D | input: | output: | (None, 2, 128) | (None, 50, 121)
conv1d_13 | Conv1D | input: | output: | (None, 50, 121) | (None, 50, 114)
conv1d_14 | Conv1D | input: | output: | (None, 50, 114) | (None, 50, 107)
concatenate_5 | Concatenate | input: | output: | [(None, 50, 121), (None, 50, 107)] | (None, 50, 228)
lstm_8 | LSTM | input: | output: | (None, 50, 228) | (None, 50, 64)
lstm_9 | LSTM | input: | output: | (None, 50, 64) | (None, 128)
dense | Dense | input: | output: | (None, 128) | (None, 10)

Figure 8: ClDNN Architecture

```python
def create_CLDNN(in_shape, dr=0.6, learning_rate=0.001):
    input = keras.Input(shape=in_shape)
    x = keras.layers.Conv1D(50, kernel_size=8,
        padding="valid",data_format = 'channels_first',
        kernel_initializer="glorot_uniform")(input)
    x = Dropout(dr)(x)
    a = keras.layers.Conv1D(50, kernel_size=8,
        padding="valid",data_format = 'channels_first',
        kernel_initializer="glorot_uniform")(x)
    a = Dropout(dr)(a)
    a = keras.layers.Conv1D(50, kernel_size=8,
        padding="valid",data_format = 'channels_first',
        kernel_initializer="glorot_uniform")(a)
    a = Dropout(dr)(a)
    c = keras.layers.concatenate([x, a])
    x = keras.layers.LSTM(64, return_sequences=True,
        kernel_initializer="glorot_uniform")(c)
    x = keras.layers.LSTM(128, dropout=dr,
        kernel_initializer="glorot_uniform")(x)
    x = keras.layers.Dense(10,activation='softmax', name="dense",
        kernel_initializer="he_normal")(x)

    optimizer = Adam(learning_rate=learning_rate)
    model = keras.Model(input, x)
    model.compile(loss='categorical_crossentropy',
        optimizer=optimizer, metrics=["accuracy"])
    print(model.summary())
    tf.keras.utils.plot_model(model, "CLDNN model 1.png",
        show_shapes=True)
    return model
```

4 Results and Analysis

The CLDNN performed best with an accuracy of 89.7% at SNR 18 with feature combination 2 and overall accuracy of 61.8% with feature combination 2. Follows it is the third CNN model implemented, with highest accuracy recorded at 85.2% at SNR 4 with the raw features and an accuracy of 58.3% overall. The RNN and LSTM models performed poorly on their own but greatly boosted the model when it had a convolutional network as a residual network.
The following figures show some plots and confusion matrices associated with different models and feature combinations to illustrate the results.
They illustrate several points:

- Performance is very poor at lower SNR values, as expected.
- Performance plateaus usually around SNR 4, although it varies slightly between one model and another.
- At higher SNR values, the most confusing classes were QAM-16 and AM-DSB.

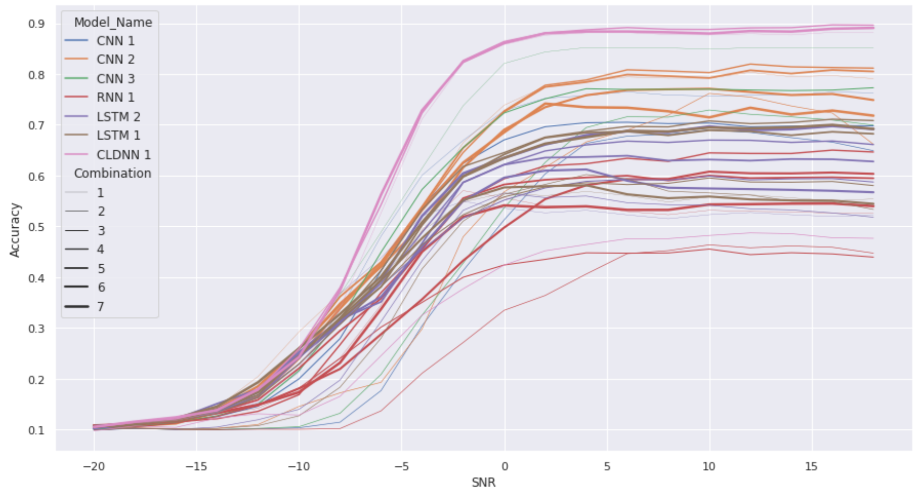


Figure 9: Comparison of all models

4.1 Accuracy against SNR plots

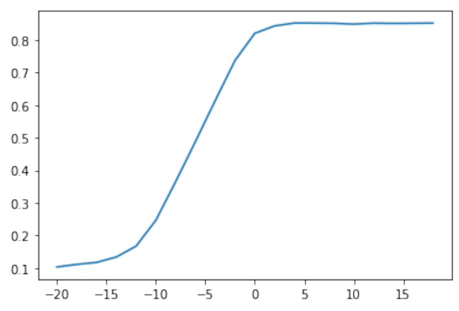


Figure 10: CNN Model 3 SNR plot

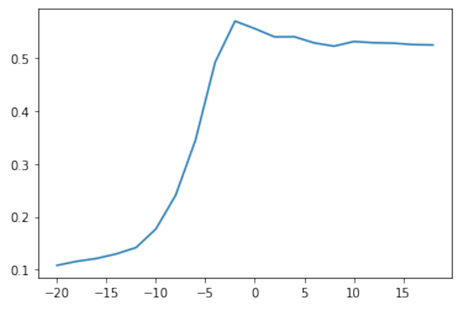


Figure 11: RNN Model 1 SNR plot

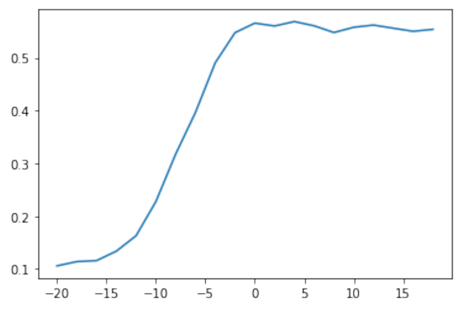


Figure 12: LSTM Model 1 SNR plot

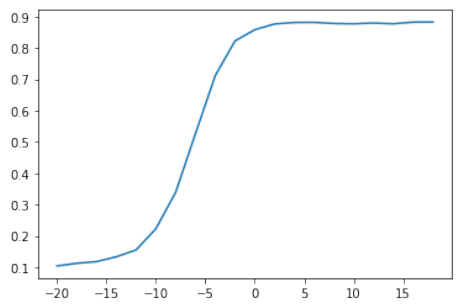


Figure 13: CLDNN Model 1 SNR plot

4.2 Confusion Matrices

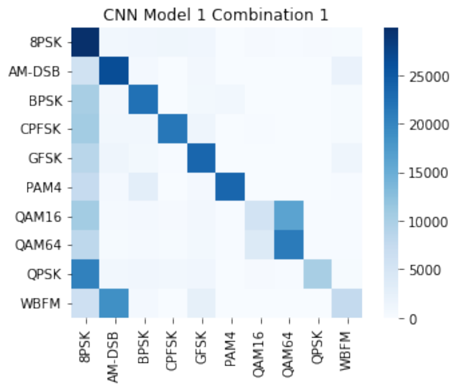


Figure 14: CNN Model 1 Confusion Matrix

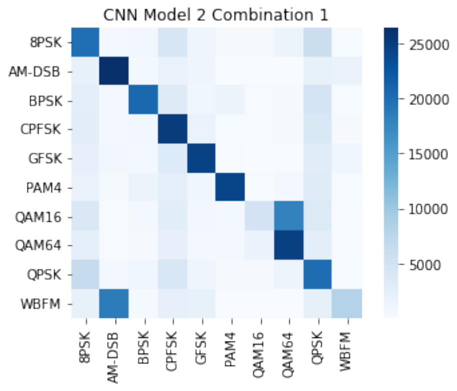


Figure 15: CNN Model 2 Confusion Matrix

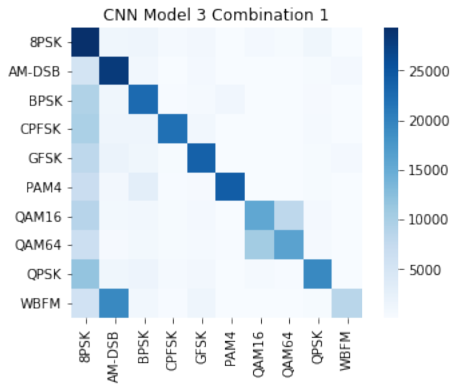


Figure 16: CNN Model 3 Confusion Matrix

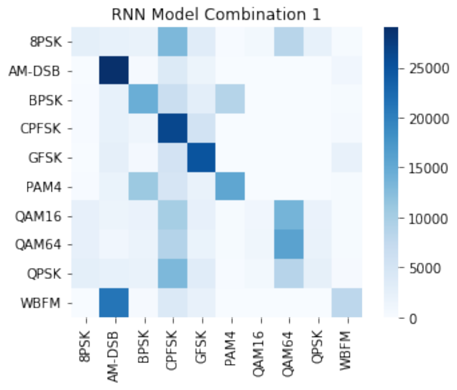


Figure 17: RNN Model 1 Confusion Matrix

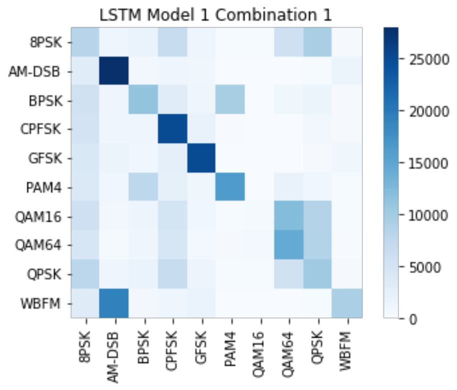


Figure 18: LSTM Model 1 Confusion Matrix

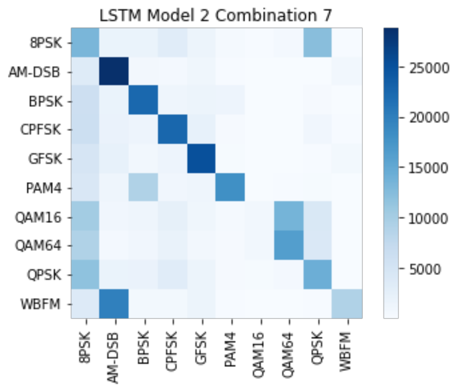


Figure 19: LSTM Model 2 Confusion Matrix

5 Improvements

5.1 Hyperparamter Tuning and GridSearchCV

An attempt was made to use the Keras GridSearchCV function to find the best values for feature combination, dropout rate, and learning rate. However, it was not possible due to limited compute resources. A pipeline was fed both the feature preprocessing pipeline as well as the model, and different values for the parameters were added to a dictionary of lists that gets passed to the function. The use of a Keras Classifier Wrapper was necessary for this step. Cross validation was done over 3 folds, and an attempt was made to run the function at a very low number of epochs, ie. 6.

A couple of points to note: it was not reasonable to try and choose the best feature space using GridSearchCV. Feature spaces would have to be recomputed and saved in the RAM without clearing it, which is not feasible with our limited resources. Moreover, trying to fine tune the dropout rate using GridSearch is unintuitive, as it functions as a way to reduce overfitting of the data. And indeed, the CLDNN model benefitted greatly from it as it was overfitting the data in the beginning. Learning rate was tested out manually and while 0.0001 was too low, with the models converging really slowly, 0.001 was perfect.

5.2 Optimizers, Kernel Initializers, Early Stop Callbacks, and Checkpoints

Throughout all models, Adam was used learning rate optimizer. An imporvement can be made to also use a scheduler in tandem with it.

Kernel Initializers such as Glorot Uniform (also called Xavier) and HeNormal were used, with HeNormal usually reserved for dense layers. RNN and LSTM models did not make use of them, however.

Early Stop callback was performed using the validation loss metric, while the checkpoints were taken using validation accuracy.

References

[1] Timothy J. O'Shea Nathan E. West. Deep architectures for modulation recognition. *arXiv:1703.09197*, Mar 2017.

[2] T. Charles Clancy Timothy J O'Shea, Johnathan Corgan. Convolutional radio modulation recognition networks. *arXiv:1602.04105*, Feb 2016.