EE 132A Final Project: LLR Net

Jonathan Nguyen

March 2020

Contents

1 Relevant Files

File Name	Purpose
EE_132A_LLR_Net_Report.pdf	Report Document (this file)
Predicting LLRs With Machine Learning.pptx	Presentation File (contains more detailed results)
LLR_NN.ipynb	Code to create the Neural Network(s)
generate_data.m	Generate noisy symbols and LLRs for a single SNR
generate_data_various_SNR.m	Generate noisy symbols and LLRs for a various SNR
generate_data_fading_single_SNR.m	Generate faded symbols and LLRs for single SNR
calculate_LLR_full_precision.m	Calculate LLR of noisy symbol vector
visualize_noise_dist.m	Create 3D plot of conditional noise distributions

2 Introduction

This project can be split up into two main components: the digital communications portion and the deep learning portion. On one hand, the digital communications side of the project involves generating symbols based off a given modulation scheme, adding noise to that symbol, and calculating the Log-Likelihood Ratio (LLR) of each bit. On the other hand, a neural network is created in order to attempt to predict these LLRs only given the noisy constellation. Note that all the digital communications functions are implemented using MATLAB and are fed into Python using .csv files.

3 Digital Communications Implementation

3.1 Generating Data

To create a random symbol vector, I utilized the randi() function in MATLAB to generate a vector with values in the range [1,M] where M is the modulation order. Then, using a pre-defined constellation map, each entry of this vector is mapped to the correct constellation. After this, complex AWGN noise is added and then we calculate the LLRs of each bit. Using three different MATLAB functions I implemented, I generated constellations and LLRS for:

1. A single SNR channel
2. A channel operating over a range of SNRs
3. A channel operating over a single SNR which also incorporates fading. This fading affects the amplitude based off a parameter I call fade_var and the phase based off a uniform distribution over $[0, 2\pi]$

3.2 Calculating LLRs

The Log-Likelihood Ratio of a bit i is defined by the equation below where b_i is the original bit sent and r is the input to the receiver:

$$LLR_{b_i} = \log\left(\frac{p(b_i = 0|r)}{p(b_i = 1|r)}\right)$$

Applying Baye's Rule to this expression, we arrive at:

$$\begin{aligned} LLR_{b_i} &= \log\left(\frac{\frac{p(r|b_i=0)p(b_i=0)}{p(r)}}{\frac{p(r|b_i=1)p(b_i=1)}{p(r)}}\right) \\ &= \log\left(\frac{p(r|b_i = 0)p(b_i = 0)}{p(r|b_i = 1)p(b_i = 1)}\right) \\ &= \log\left(\frac{p(r|b_i = 0)}{p(r|x = 1)}\right) + \log\left(\frac{p(b_i = 0)}{p(b_i = 1)}\right) \end{aligned}$$

If we assume our input bits are equally distributed (ie: $p(b_i = 0) = p(b_i = 1) = 0.5$), the second term vanishes.

$$LLR_{b_i} = \log\left(\frac{p(r|b_i = 0)}{p(r|b_i = 1)}\right)$$

Additionally, we assume our noise to be iid multi-dimensional Gaussian random variables with parameters $\mu = \mathbf{0}$ and $\Sigma = (N_0/2)I_k$. Lastly, note that there may be several constellations that correspond to $b_i = 0$. Therefore, in order to obtain $p(r|b_i = 0)$, we apply the law of total probability and sum over the subset of constellations which satisfy $b_i = 0$, denoted below as $\mathcal{X}_0$

$$p(r|b_i = 0) = \sum_{x \in \mathcal{X}_0} \frac{1}{\sqrt{N_0 \pi}} \exp\left(-\frac{1}{N_0} \|r - x\|^2\right)$$

Bringing this all together, we obtain the final expression for the LLR of bit i:

$$LLR_{b_i} = \log\left(\frac{\sum_{x \in \mathcal{X}_0} \exp\left(-\frac{1}{N_0} \|r - x\|^2\right)}{\sum_{x \in \mathcal{X}_1} \exp\left(-\frac{1}{N_0} \|r - x\|^2\right)}\right)$$

For BSPK modulation with one dimensional noise, this simplifies to the following:

$$\begin{aligned} LLR_{b_i} &= \log\left(\frac{\exp\left(-\frac{1}{N_0}\left(r - \sqrt{E_b}\right)^2\right)}{\exp\left(-\frac{1}{N_0}\left(r + \sqrt{E_b}\right)^2\right)}\right) \\ &= 4\left(\frac{E_b}{N_0}\right) r \end{aligned}$$

3.3 LLR Calculation in MATLAB

In the MATLAB code, I purposefully created my constellation maps such that each successive entry corresponded to a constellation with a binary value one greater than the last. For instance, if constellation_map[1] corresponded to the binary value 00, constellation_map[2] corresponded to 01 and so on. By doing so, figuring out which constellations have a certain bit equal to 0 or 1 becomes much easier. For example, if we looped through every constellation, we would switch whether we are looking at bit 0 or 1 with every iteration. For the second LSB, we would switch whether we are looking at bit 0 or 1 every 2 iterations. For the third LSB, we would switch every 4 iterations and so on.

```
% Calculate LLR for every bit
       for j=1:m

           % Reset num_or_dem, this variable is used to determine whether we are
               calculating probability of bit 0 or 1 (corresponding to +1 and -1
               respectively)
           num_or_dem = 1;
           num_sum = 0;
           dem_sum = 0;

           % Loop through all symbols now, note that we apply the logarithm at the end
           for k=1:num_symbols

               % Looking at 0 bit
               if(num_or_dem == 1)
                   num_sum = num_sum + exp(-1/(2*noise_var) * (abs(symbol_vec(i) -
                       constellation_map(k)))^2);

               % Looking at 1 bit
               else
                   dem_sum = dem_sum + exp(-1/(2*noise_var) * (abs(symbol_vec(i) -
                       constellation_map(k)))^2);
               end
```

```
            % Swap whether we are looking at a 0 bit or 1 bit
            % For LSB, swap every symbol
            % For 2nd LSB, swap every 2 symbols
            % For nth LSB, swap every 2^(n-1) symbols
            if(mod(k, 2^(j-1)) == 0)
                num_or_dem = num_or_dem * -1;
            end
        end

        llr_vec(j,i) = log(num_sum / dem_sum);
    end
```

4 Deep Learning Implementation

4.1 Setup and Model Architecture

After generating symbols and LLRs in MATLAB, they are exported as .csv files to be imported into Python. I used the Keras library to construct a fully connected Multiple Input, Multiple Output (MIMO) network. The inputs are of dimension 2 and correspond to the in-phase and quadrature component of our received symbol respectively. Each output corresponds to an LLR of a bit. For BPSK, there is only one output. For 16-QAM, there are 4 outputs. Note that each output shares 6 hidden layers with the other outputs while having one additional layer that is unique to it (in the figure below, these unique layers are bit_layer_i). This was done such that the outputs can learn patterns in the constellation. Below is a summary of a model used to predict 16-QAM:

Layer (type)	Output Shape	Param #	Connected to
model_input (InputLayer)	[(None, 2)]	0	
dense_39 (Dense)	(None, 128)	384	model_input[0][0]
dense_40 (Dense)	(None, 64)	8256	dense_39[0][0]
dense_41 (Dense)	(None, 32)	2080	dense_40[0][0]
dense_42 (Dense)	(None, 16)	528	dense_41[0][0]
dense_43 (Dense)	(None, 8)	136	dense_42[0][0]
dense_44 (Dense)	(None, 8)	72	dense_43[0][0]
bit_layer_1 (Dense)	(None, 4)	36	dense_44[0][0]
bit_layer_2 (Dense)	(None, 4)	36	dense_44[0][0]
bit_layer_3 (Dense)	(None, 4)	36	dense_44[0][0]
bit_layer_4 (Dense)	(None, 4)	36	dense_44[0][0]
LLR_bit_1 (Dense)	(None, 1)	5	bit_layer_1[0][0]
LLR_bit_2 (Dense)	(None, 1)	5	bit_layer_2[0][0]
LLR_bit_3 (Dense)	(None, 1)	5	bit_layer_3[0][0]
LLR_bit_4 (Dense)	(None, 1)	5	bit_layer_4[0][0]

Total params: 11,620
Trainable params: 11,620
Non-trainable params: 0

Figure 1: Model Summary for 16_QAM

4.2 Hyper-parameter Choice

4.2.1 Metric

The metric was chosen as Mean Squared Error (MSE). We are working with soft outputs. Therefore, we want to minimize the distance between the predicted LLR and the actual LLR.

4.2.2 Loss Function

Loss function was chosen to be log(cosh()). For small differences, logcosh behaves similarly to mean squared error ($\frac{1}{2}x^2$). For larger differences, it behaves closer to $\text{abs}(x) - \log(2)$ which is more linear. This has the effect of not penalizing wildly incorrect predictions as heavily which may skew the model towards over-fitting those points.

4.2.3 Activation Function

The ReLU function was chosen for the activation function empirically.Through running several tests, it was shown to result in the lowest MSE and training time compared to sigmoid, tanh, etc.

4.2.4 Batch Normalization and Dropout

I did not use Batch Normalization in my model because it resulted in worse training and validation accuracy. Perhaps if I ran my model for more epochs, this issue may have gone away. The purpose of bathnorm is to ensure that the input to each layer has the same statistics (mean and variance). This is a form of regularization which can help the model train faster and prevent overfitting.

Dropout did not help when training, so I did not use it in any model except for the fading channel. In fact, for dropout rates 0.1, 0.2, and 0.5, the performance of the models trained on a single SNR and an SNR range degraded. Even on the fading channel, the improvement was marginal. Dropout prevents overfitting by only giving the model access to a subset of the data at a time. This can be viewed as an approximate form of bagging.

4.3 Neural Network results for 16_QAM (other modulations and their losses can be found in the PowerPoint attached)

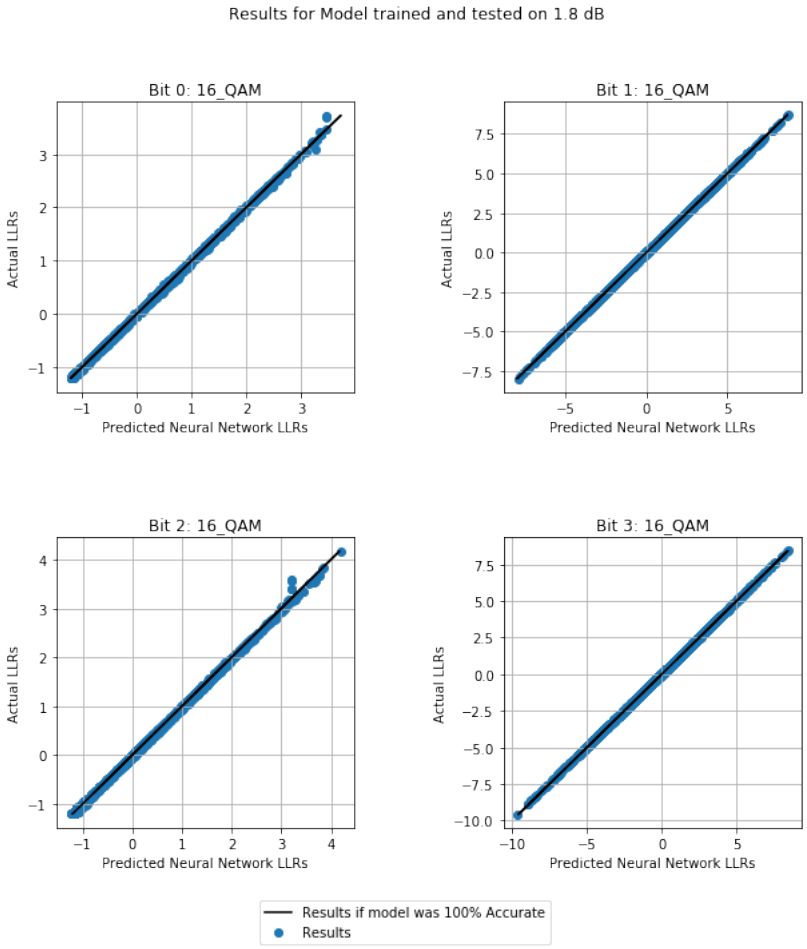

Figure 2: 16_QAM Neural Net Trained and Tested at 1.8 dB

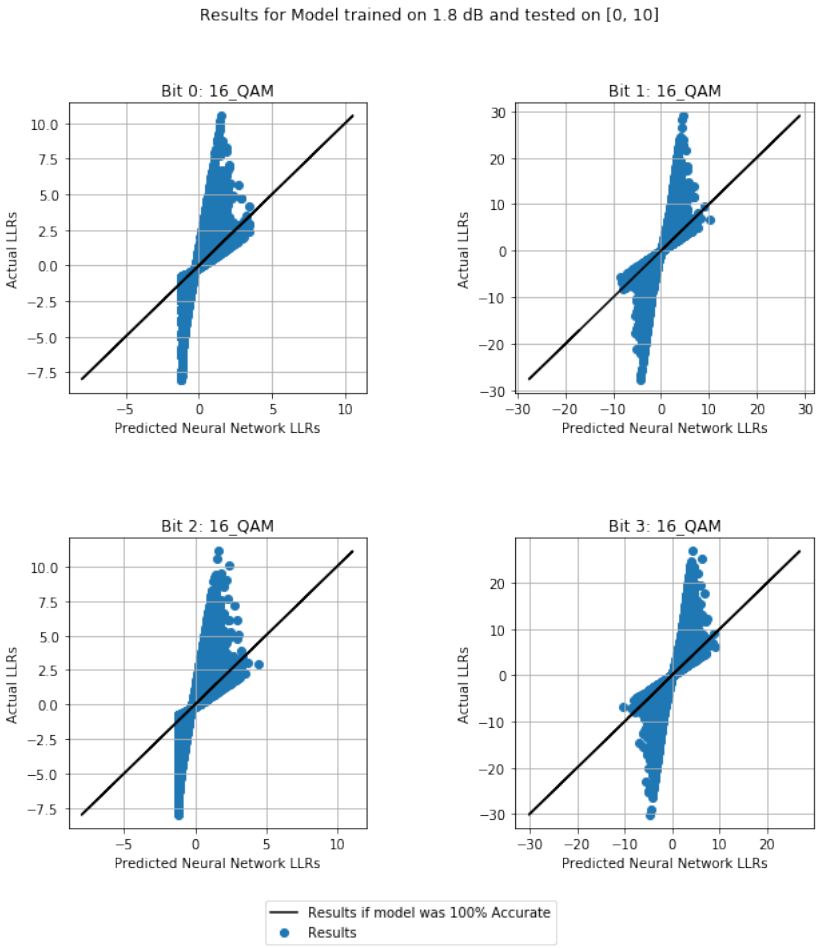

Figure 3: 16_QAM Neural Net Trained at 1.8 dB and Tested on $[0, 10]$

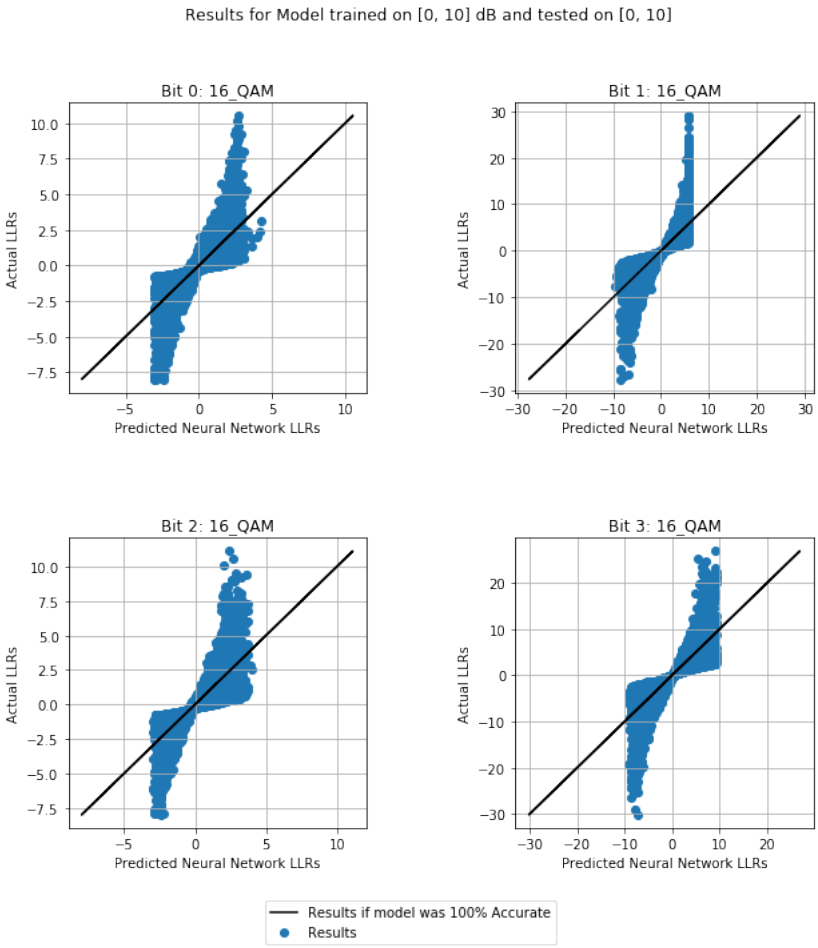

Figure 4: 16_QAM Neural Net Trained and Tested on $[0, 10]$

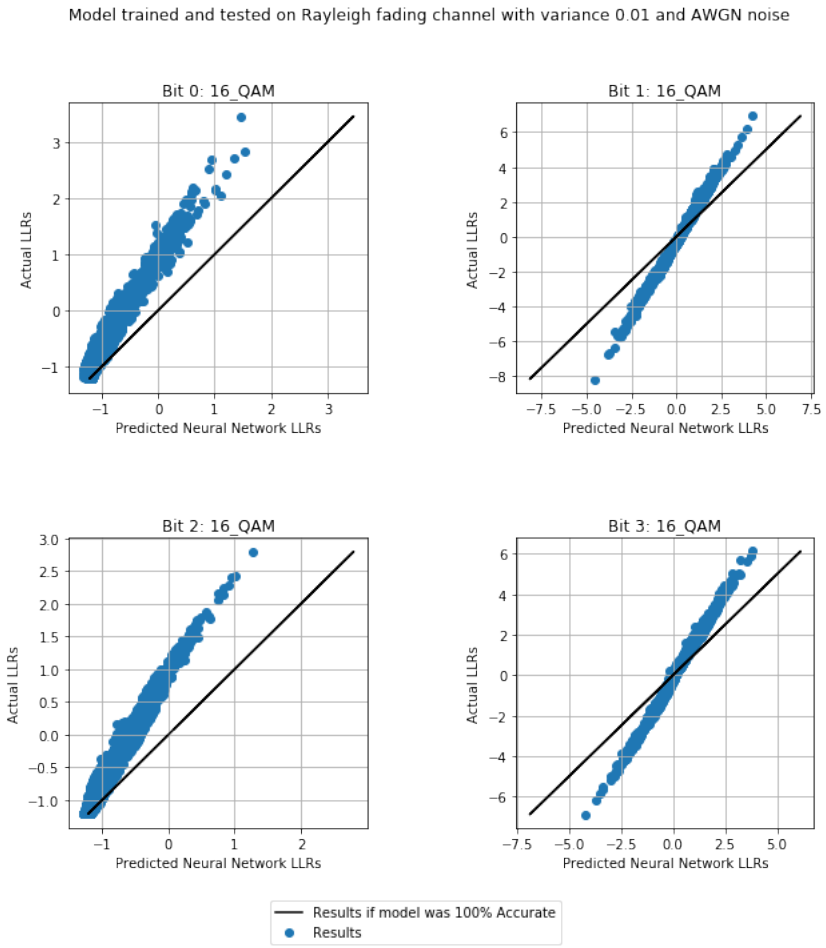

Figure 5: 16_QAM Neural Net Trained at 1.8 dB on Fading Channel

4.4 Inferences

All in all, this data seems to suggest, for a non-fading channel with AWGN noise, a fully connected neural network can perform exceptionally well for predicting LLRs given just the in-phase and quadrature component of a symbol. This approach may be extended to range of SNRs but these SNRs must be close to the ones the model was trained at. As for fading, the model as it currently stands is somewhat effective at predicting LLRs, regardless of what AWGN SNR the model was trained at. The model performs better for lower Rayleigh variances. Perhaps in the future, a more complex model can be implemented to address this.

5 Channel Coding

In order to test how effective the Neural Network generated LLRs are in comparison to the true values, I fed them into an LDPC message passing simulation. I used the IEEE 802.11n Rate 1/2 LDPC code with n=1296 with double precision belief propagation. The neural network used was trained for BPSK modulation at SNR 1.8 dB. The results are shown below.

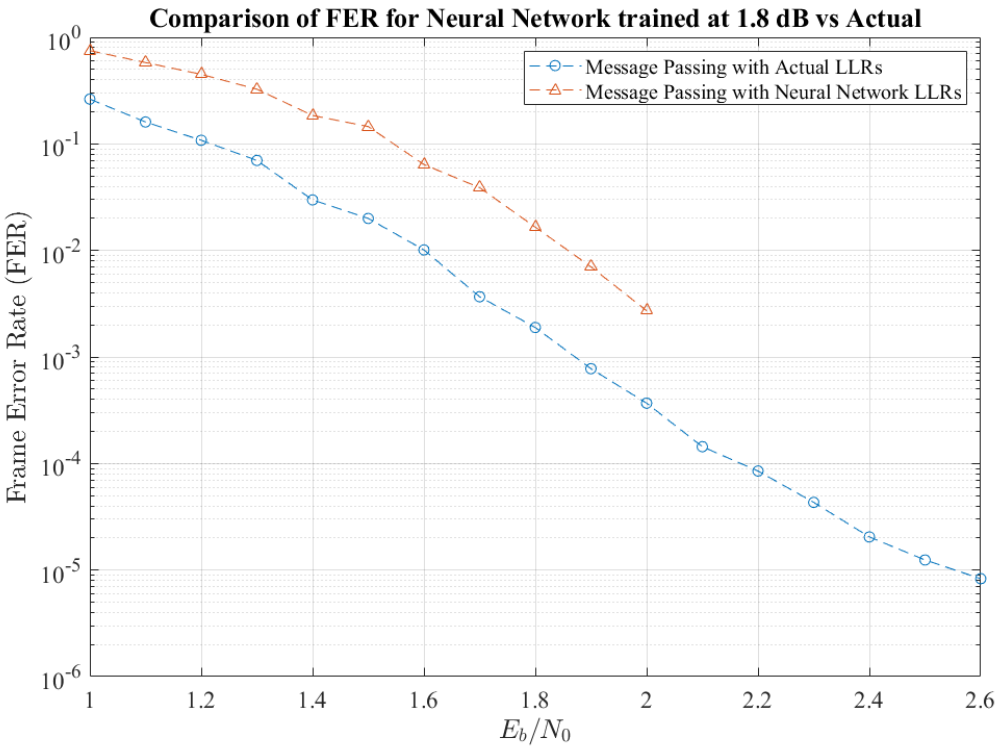

Figure 6: Frame Error Rates using actual and Neural Network generated LLRs

5.1 Inferences

The FER performance using the Neural Network generated LLRs is worse than the actual LLR, even at the SNR the model was trained at. This is most likely due to the fact that the model was not trained with channel coding in mind. As a result of channel coding, there are is less energy per transmitted bit causing a higher variance than what the model was trained at. This is most likely the cause of the discrepancy around 1.8 dB.