Iterated BP-CNN ECEN 446 Final Project

Alejandro Gomez-Leos, Giaan Nguyen

Background

Background

In “An Iterative BP-CNN Architecture for Channel Decoding”, a new system is proposed for channel decoding using low-density parity-check (LDPC) codes and belief propagation (BP), with the novel addition of a feedback convolution neural network. Over several iterations, the convolutional neural network (CNN) is able to improve the BP decoders performance.

It has been shown that LDPC codes in a white gaussian noise (WGN) channel can approach the shannon channel capacity. However, in several real-world applications, the noise signal has varying power spectral density over frequencies (unlike WGN). Since the type of noise introduced into a communication system has paramount influence over the effectiveness of the reciever, the authors proposed utilizing a convolution neural network to characterize the noise itself from the training data set.

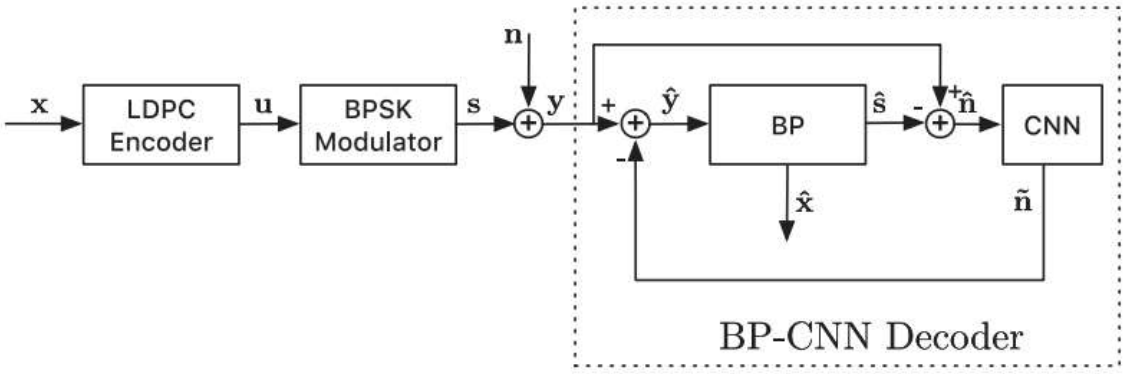

Figure 1: BP-CNN Architecture for Channel Decoding Block Diagram

At high-level, a set of K input bits **x** are extended to N bits **u**, digitally modulated, and then summed with colored noise, which is determined by parameter η generated by the attached MATLAB script. Vector **y** is the noisy vector. $\hat{s}$ is taken to be the BP module's estimation of the original message and $\hat{n}$ is computed by subtracting this estimation from y. $\hat{n}$ is the estimated noise. But since there are errors in this estimation, $\hat{n}$ is actually the original noise vector plus some error vector ξ. Through training, high ξ is designed to suffer the CNN and further train it to obtain a characterization of the channel noise such that the BP decoder's performance improves with iteration.

Several parameters may be tweaked in the provided code.

Parameter η is used to standardize the generation of the noise's covariance matrix. Higher η corresponds to more correlated noise. This feature is attractive in evaluation since it can be used to analyze the architecture's efficiency in learning the characteristics of a particular channel noise.

Normality parameter λ is coined as a "hyper-parameter" due to its important influence over BP-CNN's performance, as well as its analytic difficulty. Stemming from the loss function, we followed the authors suggestion to set this parameter based on trial and error.

We set out to evaluate channel correlation mismatch, channel noise level mismatch, and compare two cascaded BP-CNN-BP architectures.

Task 1: Channel Correlation Mismatch

Process

I opted to stay with the default CNN architecture and SNR = {0, 0.5, 1, 1.5, 2, 2.5, 3}.

For task 1, two correlation parameters η were chosen: 0.95 and 0.6. For each η the process was, as follows:

1. Set self.corr_para to η = β (0.95 or 0.6) in Configrations.py.
2. Run GenData, Train, and Simulate to "teach" the CNN to characterize the noise signal of parameter η = β and validate against a noise signal of same type.
3. For every other value γ ∈ {0.2, 0.4, 0.6, 0.7, 0.9, 0.95} – { β }, self.corr_para_simu was set to η = γ, and self.cov_1_2_file_simu was changed to search for the file pertaining to η = γ.

```
# noise information
self.blk_len = self.N_code
self.corr_para = 0.6  # correlation parameters of the colored noise
self.corr_para_simu = 0.2  # correlation parameters for simulation, this should be equal to
self.cov_1_2_file = format('./Noise/cov_1_2_corr_para%.2f.dat'% self.corr_para)
self.cov_1_2_file_simu = format('./Noise/cov_1_2_corr_para%.2f.dat'% self.corr_para_simu)
```

Figure 2: Configrations.py Changes (β = 0.6, γ = 0.2)

4. Run GenData and Simulate to generate output files in /model describing the performance of the architecture using a CNN trained with η = β. These outputs describe performance robustness checked against a signal with η = γ.
5. Steps 3-4 are repeated for all combinations of β, γ.

Challenges

Originally, 3 trainings and 15 runs had been completed with faulty data. The source of the issue was that self.corr_para_sim_file_simu's default setting seeks the training η = β covariance file instead of the test η = γ covariance file. Due to this, the outputs of each simulation resulted in the same exact data from run to run.

Once it was determined that this line was the source of the error, data collection continued uninterrupted by using the above process.

Results

The first η chosen to train the CNN was η = 0.95. This was tested for η = 0.2, 0.4, 0.6, and 0.9.

For fixed SNR ∈ { 0, 0.5, 1, 1.5, 2, 2.5, 3}, each test η was plotted against the BP-CNN architecture's resulting BER (bit-error rate) .

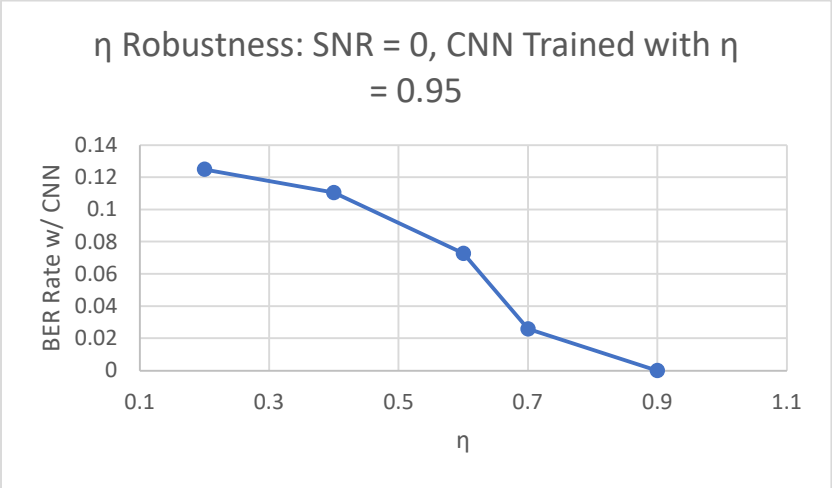

Figure 3: Robustness for SNR = 0 Case, η = 0.95 Training

As test η approaches the training η, the BER decreases substantially, with a sharp drop within ~0.3 of training η. Since this is for SNR = 0, this is the worst case scenario, when the noise is the strongest. As the SNR increases, that means the signal strength increases in proportion to noise, so it is expected that this curve will flatten out for higher SNR.

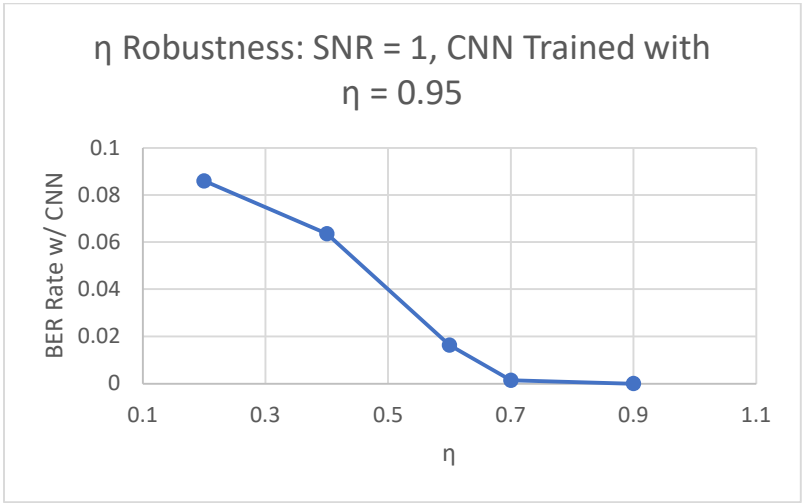

Figure 4: Robustness for SNR = 1 Case, η = 0.95 Training

It is noted at this point, that the η range from 0.4 to 0.6 has flattened out. Increasing SNR further flattens the entire curve. The trend is observable over all SNR values.

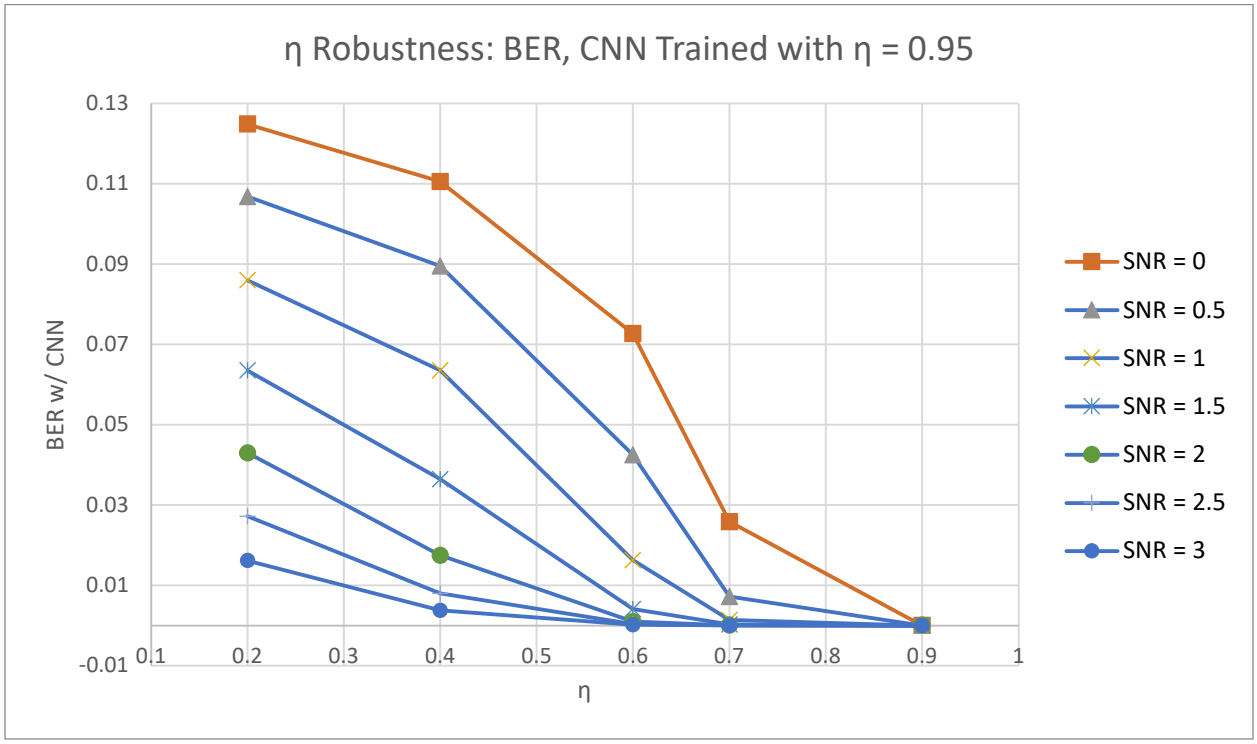

Figure 5: Robustness, All SNR, η = 0.95 Training

As expected, the CNN performs the best (lowest BER) in the channel with correlated noise with covariance parameter η = 0.95.

A surprising result was the 0.00 BER w/ CNN observed at the test η = 0.9. While low BER was expected, an all zero BER seems too perfect. However, this collection step was ran three times and all three times output an all-zero column for BER w/ CNN. Originally, I had planned to plot the performance loss dB, but the reference BER w/ CNN renders the calculation undefined (due to being zero).

Next, the CNN was trained to handle η = 0.6, and tested against other η = 0.2, 0.4, 0.7, 0.9, 0.95.

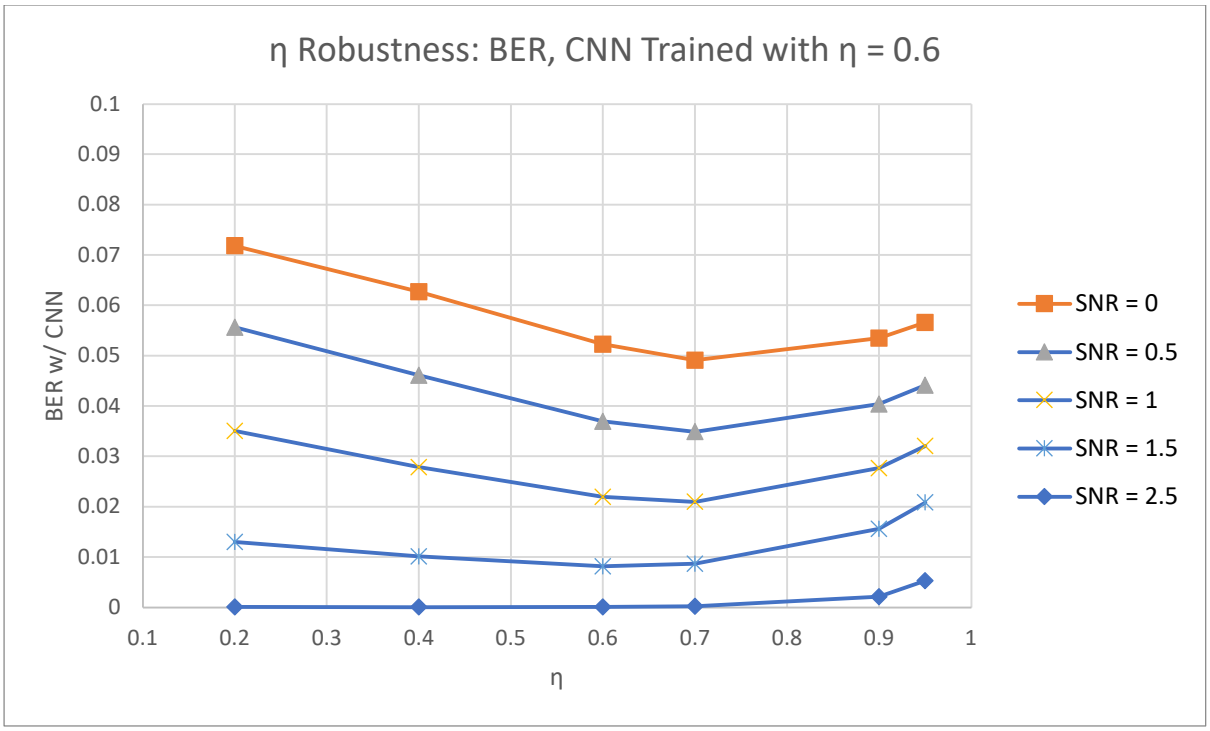

Figure 6: Robustness, All SNR, η = 0.6 Training

Oddly enough, it seemed that the CNN performed the best here under η = 0.7, for lower SNR. By observing the Performance Loss (dB) as 10*log(BER_{test} / BER_{train}), one can see that there is indeed a slight gain (negative loss) in performance at η = 0.7 (diagram shown in Figure 7).

Other than this, the expected dip in BER occurs close to the training η = 0.6

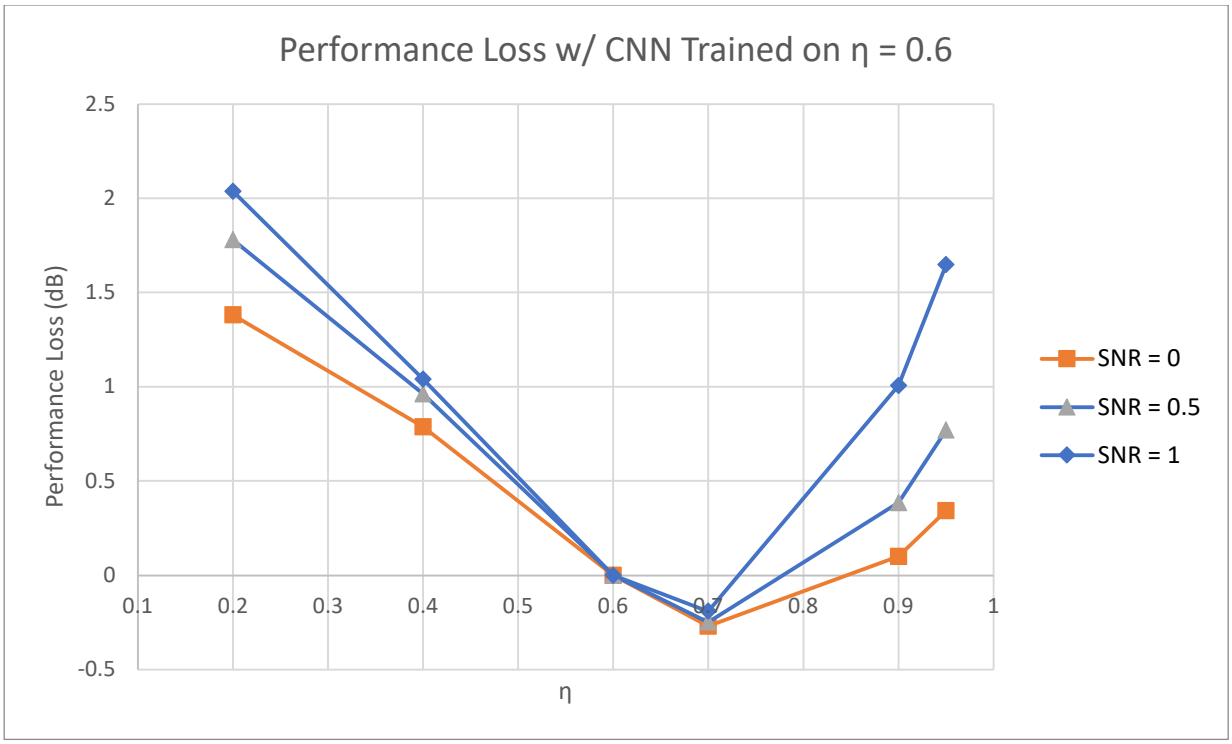

Figure 7: Robustness for SNR = 1 Case, η = 0.6

The exact cause of this was not identifiable. Note that Figure 13 of the research paper does not indicate any gains in performance.

Conclusion

As with any machine learning system, trained models tend to perform better the closer validation tasks are to training data. This is inherently the same as image recognition in cases where the model has already seen the validation images. Therefore, it is no surprise that BER is lowest when the model is confronted with correlated noise when it has been trained on similarly correlated noise.

Even with unfamiliar noise, the architecture still outperforms standard BP decoders with low bit error rates, indicated by the paper.

Task 2: Channel Noise Level Mismatch

For the second task, a SNR mismatch analysis is performed. While the CNN was trained using data generated under a single SNR value, the BP-CNN decoder during simulation would use a range of SNR values that will not include the training SNR; the BER is then computed across different training SNR values. That is, the robustness of the system to the training data – when the training data is generated under different channel conditions than that of the decoder – is of interest.

Process

While only two training SNRs were necessary, four different values were chosen for SNR: 0.25, 1.25, 2.25, and 4.25 dB. The CNN architecture was kept as the default, with noise parameter $\eta = 0.5$. The procedures are as follows:

1. Set self.SNR_set_gen_data to training SNR = β (0.25, 1.25, 2.25 or 4.25) in Configrations.py.
2. For a single training SNR, run GenData and Train modes to train the CNN.
3. Make self.eval_SNRs equal to the set {-0.5, 0.25, 0.75, 1.25, 1.75, 2.25, 2.75, 4.25} – { β }, and run Simulation mode to generate output files in /model describing the performance of the architecture.
4. Steps 2-4 are repeated for all β.

Challenges

Initially, the BER and noise outputs gave similar results, with differences within hundredths of each other. However, once the entire GenData-Train-Simulation workflow was run for each different training SNR, the results for each training SNR were unique. In short, regardless of whichever mismatch is tested, all three modes must be run in succession.

Results

Under moderate correlation η = 0.5, four different training SNR values were tested separately against a decoder SNR range specified in the process section, the results of which are found in Figure 8.

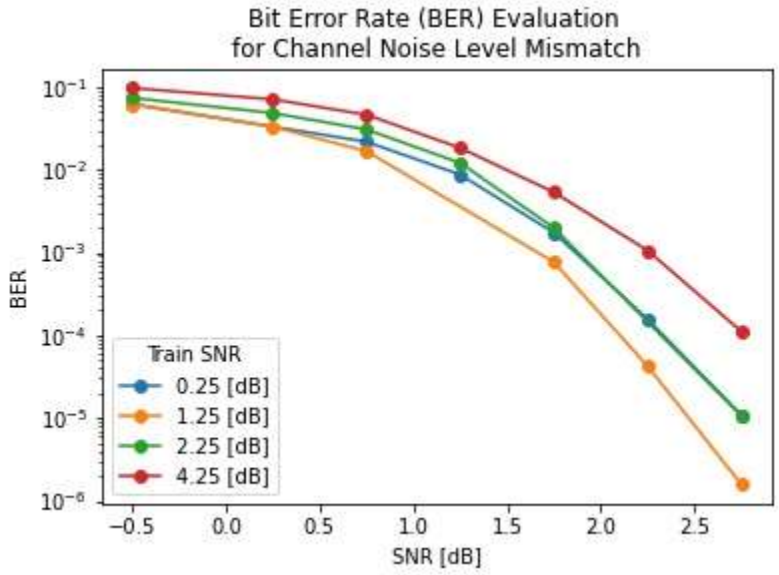

Figure 8: BER Evaluation for Channel Noise-Level Mismatch

With regards to BER, training SNR 1.25 dB gives the best performance as it has the lowest BER across all decoder SNRs, whereas train SNR 4.25 dB gives the worst as it has the highest BER across the decoder SNR range. Interestingly, train SNR 0.25 dB has a similar behavior to 1.25 dB for small decoder SNR values before it approaches the train SNR 2.25 curve.

A potential explanation at why train SNR 1.25 dB performs the best among the four may be related to the channel conditions for the decoder. Under matched conditions, the BER performance is good (i.e., when the BER curve is low) since both the CNN and the decoder utilizes the same SNR value(s). In the absence of matching, consider the average of the decoder SNR range, excluding the train SNR, as provided in Table 1. Seemingly as the train SNR is closer to the average decoder SNR, the better the BER performance is.

Since train SNR 1.25 dB is closer to the average decoder SNR than its three train counterparts, it performs better than the three. Similarly, since train SNR 4.25 dB is further away from the respective average, it performs the worst. Theoretically, choosing train SNR 1.75 should give a better performance than train SNR 1.25.

Table 1: Better Performance when Train SNR ~ Average Decoder SNR

Train SNR (dB)	Average Decoder SNR (dB)
-0.5	1.892857
0.25	1.785714
0.75	1.714286
1.25	1.642857
1.75	1.571429
2.25	1.5
2.75	1.428571
4.25	1.214286

Now consider the residual noise of the system, which is defined as the difference between the noise estimated by the CNN and the true noise. Alternatively, residual noise is a measure of how accurate the noise estimate is; the smaller the residual noise, the more accurate the estimate. As seen in Figure 9, the residual noise is the lowest for train SNR 4.25 dB; while this suggests that the noise estimate is accurate, the BER is still the highest among its counterparts.

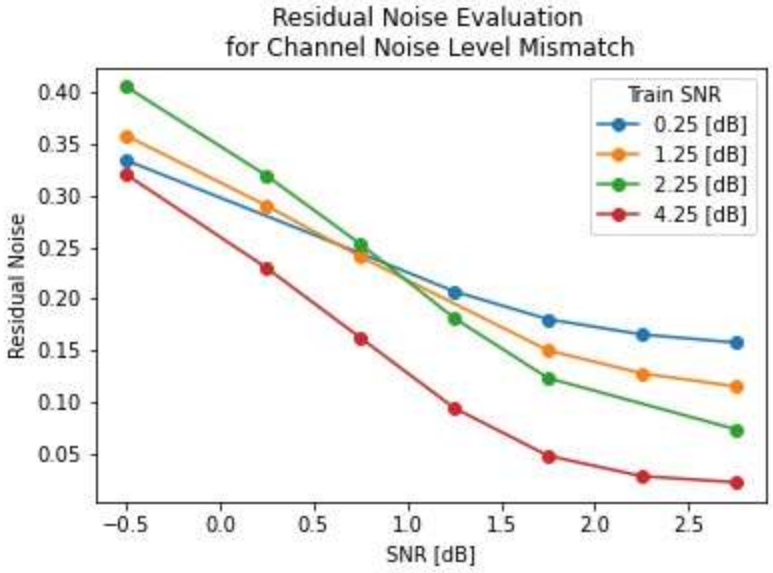

Figure 9: Residual Noise Evaluation for Channel Noise-Level Mismatch

While the residual noise curve for train SNR 2.25 dB is higher than that of 0.25 dB, interestingly the case is switched after decoder SNR of about 1.0 dB; train SNR 1.25 dB stays in the middle of both SNRs throughout, which may be an indication of 1.25 dB being a suitable candidate for mismatch with minimized error.

Conclusions

Similar to the first task, trained models seem to perform better when their characteristics resemble close to that of the validation setup. As seen with the BER curves, the closer the train SNR was to the average of the subset of decoder SNRs, the lower the BER was, and thus the better the system performs.

Task 3: Cascaded BP-CNN Architectures

For the third task, the provided code was modified to implement a BP-CNN_1-BP-CNN_2-BP architecture, where CNN_1 and CNN_2 are two CNNs with different parameters/structure. This was further evaluated against the iterative BP-CNN-BP-CNN-BP for the case where both CNNs have the same parameters.

CNN_1 was given the default parameters of $\lambda = 1$, CNN architecture {4; 9, 3, 3, 15; 64, 32, 16, 1}. This architecture was also used in the control experiment, where both CNN's have the same parameters. Note that noise parameter $\eta = 0.5$.

CNN_2 was tested with four different variations.

1. $\lambda = 0.1$, CNN architecture {4; 9, 3, 3, 15; 64, 32, 16, 1}.
2. $\lambda = 10$, CNN architecture {4; 9, 3, 3, 15; 64, 32, 16, 1}.
3. $\lambda = 1$, CNN architecture {4; 5, 3, 3, 9; 64, 32, 16, 1}.
4. $\lambda = 1$, CNN architecture {4; 7, 5, 5, 15; 64, 32, 16, 1}.

Process

To deal with the modified architecture, separate it into two stages. The first stage (BP-CNN_1-BP) generates some noise samples, trains the net on the data, and simulates using the residual noise due to the net; this is similar to the default workflow, with the exception of extracting output noise data instead of the default BER text file. Since the code for generating noise samples is similar to that for simulation, one just needs to copy code chunks from one function to the other. Starting from the default code and parameters, the first-stage procedures are as follows:

1. In Iterative_BP_CNN.py, under the function simulation_colored_noise(), comment out any lines pertaining to simulation times or BER.
2. From the generate_noise_samples(), copy any code pertaining to 'Training' or 'Test', and paste them into simulation_colored_noise(). Include generate_data_for and train_config as parameters for simulation_colored_noise().
3. In main.py, under the simulation conditional statement, define a variable save_mode that will take on either 'Training' or 'Test'. Pass the variable into the modified function simulation_colored_noise.
4. Run GenData and Train modes as usual.

5. Set save_mode = 'Training' and run simulation. Then set save_mode = 'Test' and run simulation once more. Noise samples for both training and test should be outputted and ready for the second stage.

The second stage (CNN_2-BP) takes the output noise data from the first stage, trains the new net on such data, and simulates using the residual noise. Unlike the first stage, there is no need for another GenData run, and the default code for simulation can be run such that the output is the BER text file. Starting from the default code and parameters, the second-stage procedures are as follows:

1. In Configrations.py, change self.filter_sizes and self.normality_lambda to the appropriate values of interest.
2. Run Train and Simulation modes, using the outputs from the first stage as data to train.
3. Repeat steps 1-2 for each CNN_2 setup (including the control where $CNN_2 = CNN_1$).

Challenges

The primary challenge was figuring out where and how to edit the code. Initial attempts included changing the Configrations.py file using the setting self.same_model_all_nets. However once it was understood that the architecture essentially is comprised of two stages, then it is only a means of figuring out how to extract the noise data after the second BP decoder. From main.py, one can see that both GenData and Simulation modes use the same module loaded from Iterative_BP_CNN.py. Going into said file and looking at the respective functions they call on, the functions generate_noise_samples() and simulation_colored_noise() have a similar for-loop structure; therefore, it only made sense to copy code from generate_noise_samples() to simulation_colored_noise().

Results

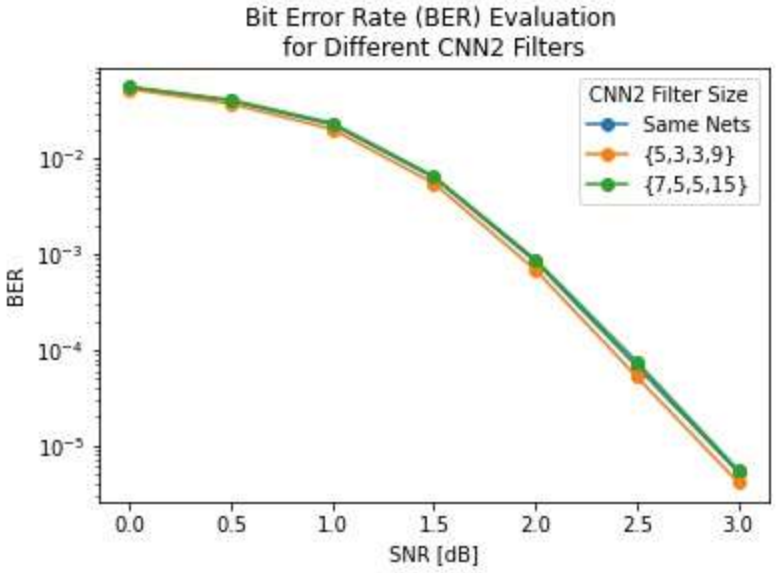

Figure 10: BER Evaluation for Different CNN_2 Filters

Using BER as the primary criteria for performance, it was observed that increasing the filter sizes resulted in the same BER, while decreased filter sizes performed marginally better compared to the same CNN control experiment. However, it is noticable that all three experiments resulted in similar bit error rates.

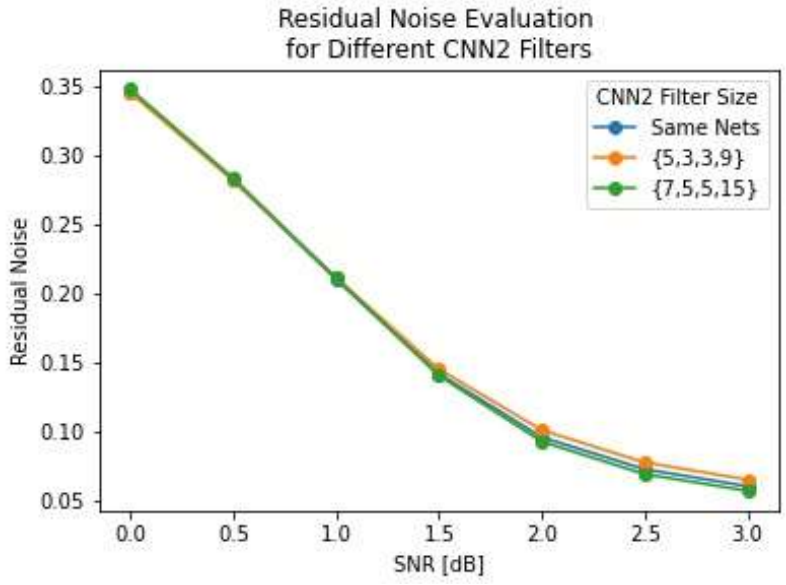

Figure 11: Residual Noise Evaluation for Different CNN_2 Filters

As mentioned earlier, residual noise may also be evaluated. For low SNR, all three experiments performed similarly. For higher SNR, the larger filter sizes resulted in lower residual noise, albeit marginally different.

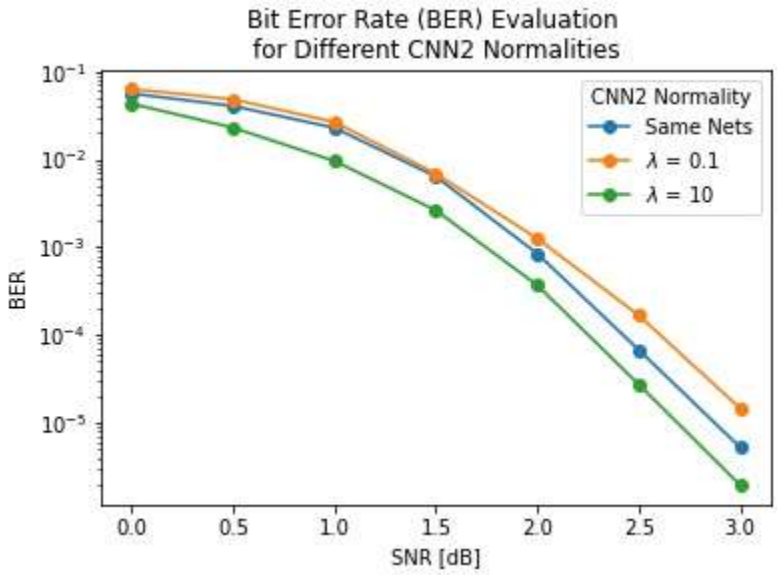

Figure 12: BER Evaluation for Different CNN_2 Normalities

By increasing the normality hyperparameter to $\lambda = 10$, it outperformed the control CNN in terms of the BER metric, across all SNRs. In contrast, when decreasing the hyperparameter to 0.1, it underperformed in comparison to the other two counterparts.

Again, these performances were evaluated in terms of residual noise.

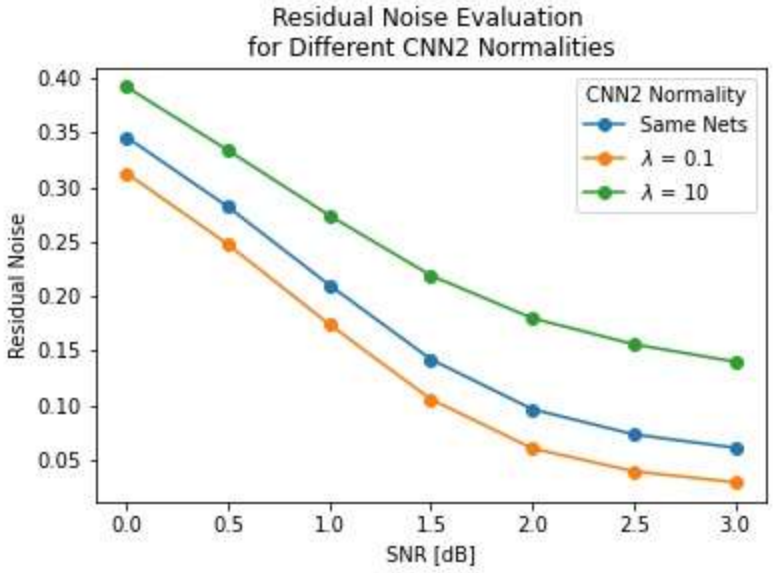

Figure 13: BER Evaluation for Different CNN_2 Normalities

Surprisingly, the CNN_2 trained with $\lambda = 0.1$ achieved the smallest residual noise, surpassing the control CNN in performance.

Conclusion

Overall, our experimental data indicates that BP-CNN_1-BP-CNN_2-BP can surpass the BER performance of BP-CNN-BP-CNN-BP either by increasing the normality hyperparameter λ, decreasing the filter size, or potentially a combination of the two, when the correlation parameter is set to 0.5.

Contribution

Both members contributed equally to the report and presentation.