EFFECT OF NOISE INJECTION IN SEQUENTIAL MODELS

Spandan Anaokar, 210260055

Nahush Kolhe, 210260034

1 PROBLEM STATEMENT

This project focuses on exploring noise injection techniques within the training process of various neural network architectures, including RNNs, LSTMs, and transformers. Noise injection during gradient descent has gained attention for its potential to provide regularization and smoothing effects, as highlighted in the paper "Explicit Regularization in Overparametrized Models via Noise Injection."Orvieto et al. (2023) However, its effectiveness across different neural network architectures remains relatively unexplored.

To address this gap, the project aims to investigate the impact of injecting noise before computing gradient steps in Sequential Task neural networks. Specifically, the goal is to assess whether subtle perturbations introduced during training can induce explicit regularization and enhance generalization performance, particularly in large-scale RNNs, LSTMs, and transformers.

2 PREVIOUS WORK

Previous studies have extensively investigated the incorporation of noise within gradient descent to impart desirable properties such as smoothing and regularization effects in machine learning and deep learning models. Early research demonstrated the efficacy of noise injection in inducing explicit regularization for simple models based on various norms. However, as neural network architectures grew in complexity and size, challenges emerged, particularly regarding variance explosion when applying noise injection to overparametrized models.

Recent advancements in the field have proposed novel strategies to overcome these challenges. One such approach involves the implementation of independent layer-wise perturbations, which decouple perturbations across layers to mitigate variance explosion while maintaining regularization benefits. By decoupling perturbations across layers, researchers achieved explicit regularization without encountering variance explosion, even in large-scale deep learning models. These developments have laid a solid foundation for our study, which aims to extend the understanding of noise injection effects in transformer networks and explore its implications for model performance and generalization capabilities.

3 THEORETICAL BACKGROUND

NOISE INJECTED BEFORE GRADIENT DESCENT (GD):

Perturbations are introduced to the iterate before evaluating the gradient in this approach. The resulting so-perturbed gradient is then utilized to update the unperturbed iterate. Typically applied in convex optimization, this method smooths non-smooth objective functions, thereby facilitating optimization. Furthermore, in non-convex scenarios, it exhibits an implicit bias towards flat minima, which are known to generalize better, thus enhancing the optimization process.

NOISE INJECTED AFTER GRADIENT DESCENT (GD):

Perturbations are added post-gradient step in this method, accelerating the escape from spurious local minima and saddle points. By injecting noise after each GD step, this approach speeds up the exploration of the optimization landscape, aiding in quicker convergence. These methods, corresponding to discretizations of continuous-time Langevin dynamics, contribute to the diffusion coefficient, thereby enhancing the optimization process.

4 EXPERIMENTS

4.1 MAIN ALGORITHM

```
# Begin training loop
model.train()  # Set the model in training mode

for batch_idx, (data, target) in enumerate(train_loader):
# Iterate over batches in the training data

    # Model perturbation
    if settings["noise"] != "no":
    # Check if noise perturbation is enabled
        param_copy = []
        # Initialize an empty list to store copies of model parameters
        with torch.no_grad():
        # Temporarily disable gradient calculation
            i = 0
            # Initialize counter for iterating over model parameters
            for param in model.parameters():
                # Iterate over model parameters
                param_copy.append(param.data)
                # Save a copy of the parameter data
                # Check noise type
                if settings["noise"] in ["all", "after_all"]:
                    # Add noise to all parameters
                    param.data = param.data + (sigma_curr / math.sqrt(
                                                        n_groups)) *
                                                        torch.normal(0,
                                                        1, size=param.
                                                        size(), device=
                                                        device)
                elif settings["noise"] in ["layer", "after_layer"]:
                    # Add noise to parameters of selected layer
                    if i == (iter % n_groups):
                    # Check if current layer matches selected layer
                        param.data = param.data + sigma_curr * torch.
                                                            normal(0, 1,
                                                             size=param.
                                                            size(),
                                                            device=
                                                            device)
                    i += 1  # Increment counter
        # End loop over model parameters

    # Backpropagation
    data = data.to(device)  # Move input data to device (e.g., GPU)
    target = target.to(device)  # Move target labels to device
    optimizer.zero_grad()  # Reset gradients
    output = model(data)  # Forward pass: compute model predictions
    loss = criterion(output, target)  # Compute loss
    loss.backward()  # Backpropagate gradients

    # Model recovery
```

```
    if settings["noise"] != "no" and settings["noise"] != "after_all" and
                                        settings["noise"] != "
                                       after_layer":  # Check if noise
                                       perturbation was applied with
                                       torch.no_grad():
    # Temporarily disable gradient calculation
            i = 0  # Initialize counter for iterating over model
                                                parameters
            for param in model.parameters():  # Iterate over model
                                                parameters
                param.data = param_copy[i]  # Restore original parameter
                                                    data
                i += 1  # Increment counter

    optimizer.step()  # Update model parameters using optimizer
```

In our algorithm, we employ a customized version of the methodology initially proposed by the authors. This adaptation incorporates two distinct modes, each dictating the timing of noise injection relative to the gradient descent computation. The fundamental principle revolves around the replication of all model parameters prior to the introduction of noise. Subsequently, noise is superimposed onto these parameters, following which the direction of gradient descent is determined at the perturbed point.

The two divergent approaches stem from the manner in which we handle the perturbed weights. As delineated in the original literature, the first approach, termed "Before GD Noise," involves reverting to the original weights and subsequently appending the gradient descent corresponding to the displaced weights. Conversely, the alternative strategy eschews the utilization of stored weights, opting instead to execute gradient descent directly at the perturbed location. This distinction marks a pivotal divergence in the execution of our algorithm, leading to varied implications in the optimization process.

5 RESULTS

In our approach we attempt to match the results given in the paper and further apply the same method on different models. Along with this we also introduce a new baseline measure which was not explored in the paper that is the result of using After GD Descent Noise in both layerwise and all layers.
Due to lack of resources, the training could not be conducted as extensively as was done in the paper. However by reducing the number of epochs and increasing the learning rate we have managed to reach convergence while at the same time computing efficiently.

5.1 BASELINES

5.1.1 FMNIST

Fashion MNIST (FMNIST) is a dataset developed as an alternative to MNIST, specifically tailored for benchmarking machine learning algorithms in the context of fashion items recognition. Like MNIST, FMNIST comprises 28x28 grayscale images, but instead of handwritten digits, it features various fashion items such as t-shirts, trousers, dresses, and shoes. With 60,000 training images and 10,000 testing images, FMNIST offers a more challenging dataset compared to MNIST, providing a diverse set of images representing different clothing categories. Each image in FMNIST is labeled with the corresponding fashion item class, allowing for the evaluation of algorithms in fashion recognition tasks. We add noise to training on the FMNIST dataset and then compare our results with that of the paper.
In the implementation we have utilized the following: For MNISTNet1 (shallow):

- Input layer: 784 neurons (28x28)
- Hidden layers: Three fully connected layers with 500 neurons each

- Output layer: 10 neurons

For MNISTNet2 (wide):

- Input layer: 784 neurons
- Hidden layers: Three fully connected layers with 5000 neurons each
- Output layer: 10 neurons

For MNISTNet3 (deep):

- Input layer: 784 neurons
- Hidden layers: Five fully connected layers with 1000 neurons each
- Output layer: 10 neurons

5.2 KEY OBSERVATIONS

We see that the result are very similar to those described in the paper. Both without noise and layerwise noise give better results than simply adding noise to all layers.

5.2.1 CIFAR-10

CIFAR-10 is a dataset designed for object recognition tasks, containing 32x32 color images across ten distinct classes. These classes include airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks, with 6,000 images per class. CIFAR-10 provides a more complex and realistic challenge compared to MNIST and FMNIST due to its color images and diverse object categories. With 50,000 training images and 10,000 testing images, CIFAR-10 facilitates the evaluation of algorithms in real-world image recognition scenarios. Each image in the dataset is labeled with the corresponding object class, enabling researchers to train and test machine learning models for object classification tasks.

5.2.2 KEY OBSERVATIONS

The after noise methods do not provide good enough results as they completely fall off. For more bigger CNN simple SGD leads to very slow learning. Adding noise makes the process faster and gives comparative results for both layerwise and complete noise addition.

5.3 MNIST DATASET

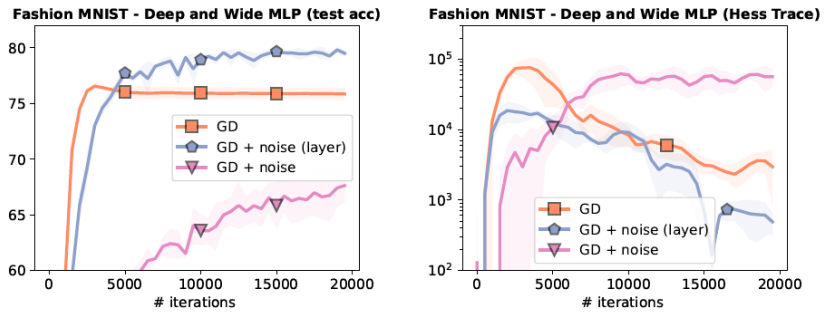
We also try out our model on the more simple MNIST dataset. The MNIST dataset, standing for Modified National Institute of Standards and Technology, is a widely recognized benchmark dataset in the field of machine learning. It comprises a collection of 28x28 grayscale images of handwritten digits ranging from 0 to 9. With a total of 60,000 training images and 10,000 testing images, MNIST serves as a fundamental resource for training and evaluating image classification algorithms. Each image in the dataset is associated with a corresponding label indicating the digit it represents. MNIST's simplicity and accessibility have made it a standard dataset for researchers and practitioners alike, providing a foundational platform for developing and testing various machine learning models.

5.3.1 KEY OBSERVATIONS

For a small neural network the various method do not have much difference in results. But when the size increases we see that layerwise noise gives us the best results followed by complete noise. the after noise methods perform the worst.

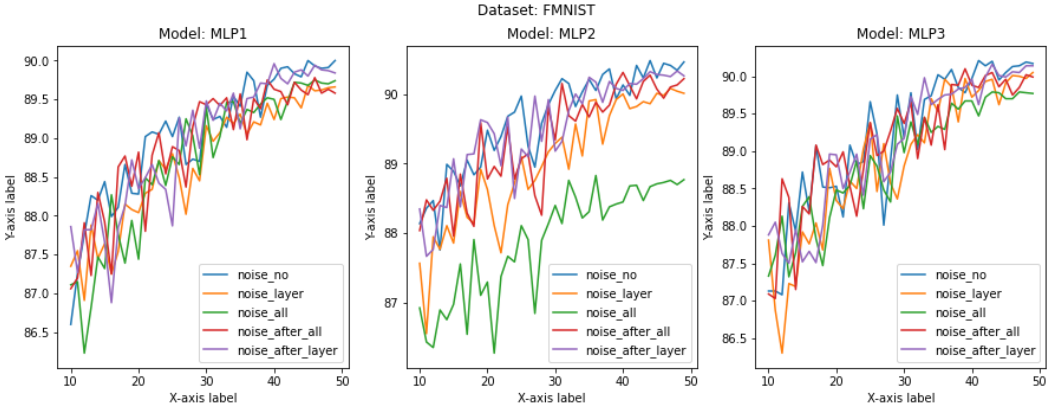
5.4 TEXT CLASSIFICATION

Text classification is a foundational task in natural language processing (NLP) that revolves around categorizing text documents into predefined classes or categories based on their content. It plays a

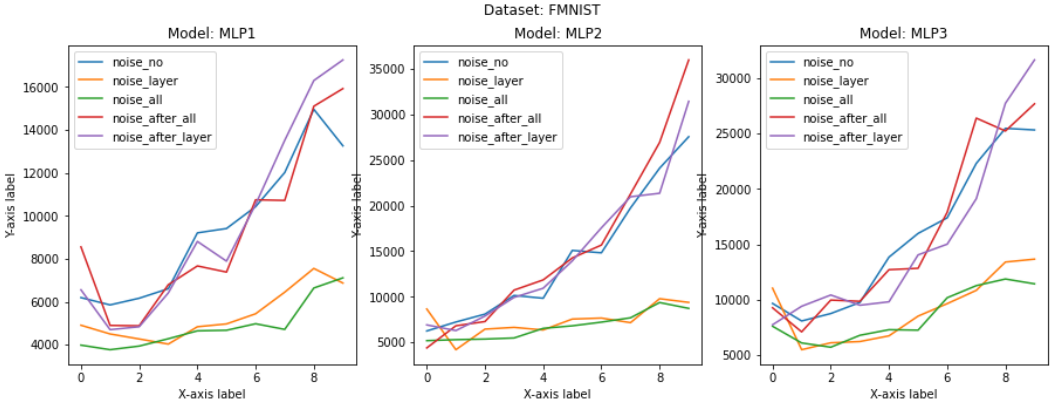


(a) FMNIST results from paper

Figure 1: FMNIST results from paper



(a) FMNIST results from our implementation: Accuracy



(b) FMIST from our implementation: Hessian

Figure 2: FMIST from our implementation

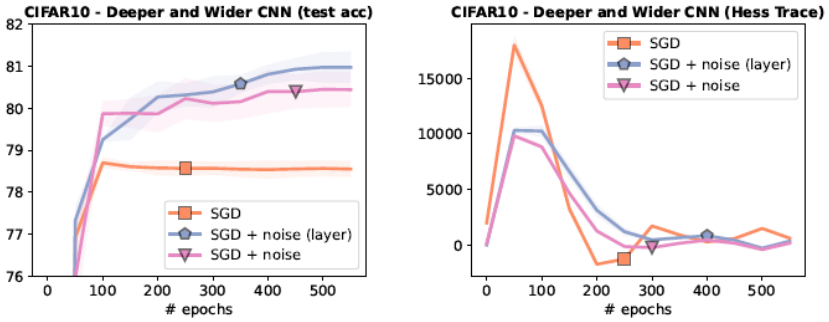


Figure 3: CIFAR results from paper

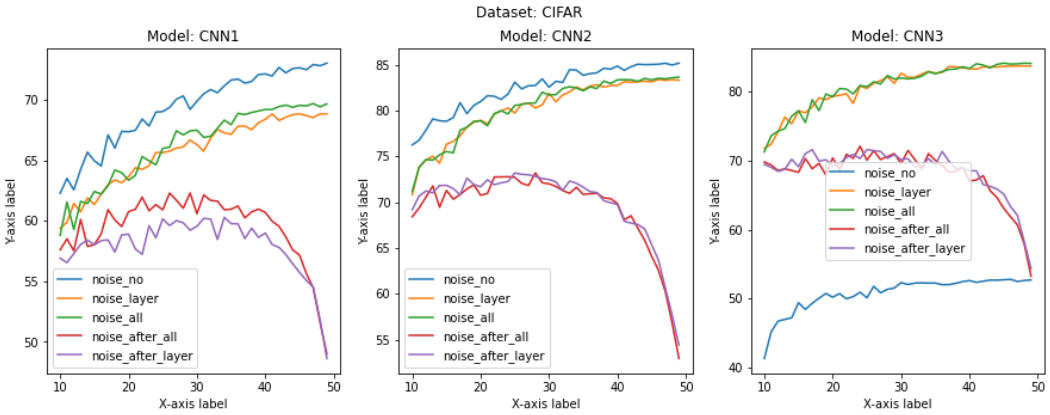
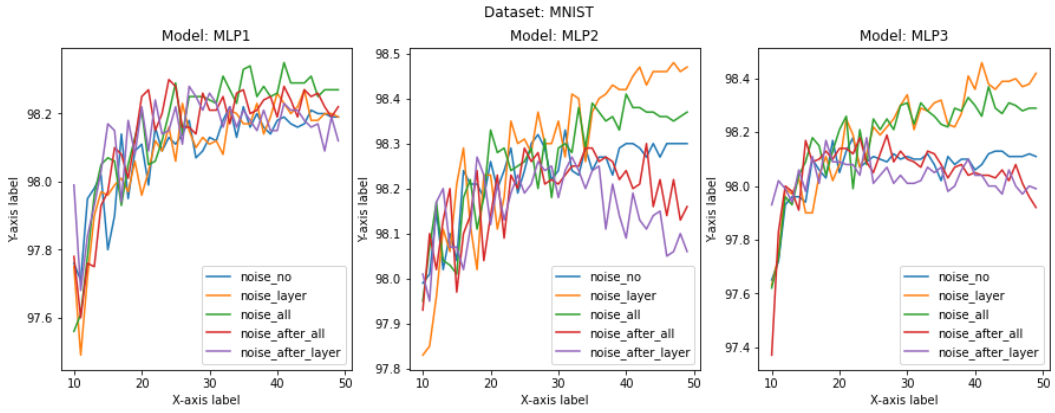
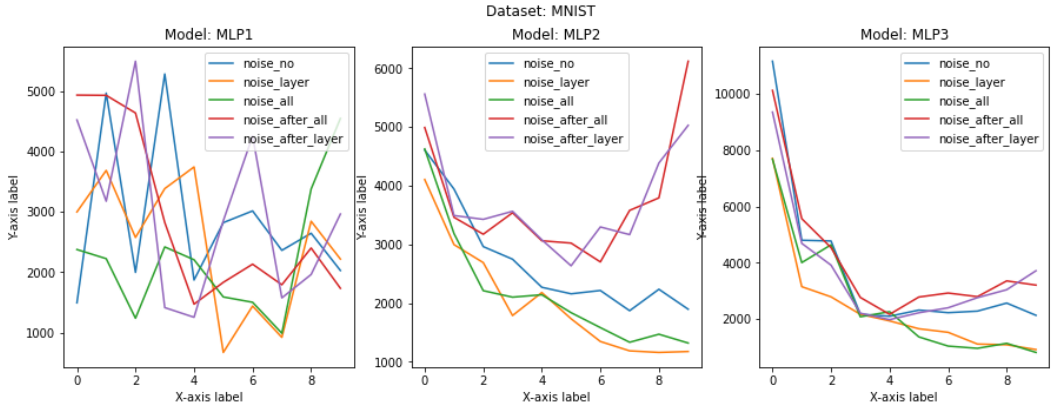


Figure 4: CIFAR results from out implementation



(a) MNIST results from our implementation: Accuracy



(b) MNIST from our implementation: Hessian

Figure 5: MNIST Results

crucial role in various NLP applications such as sentiment analysis, topic categorization, and spam detection.

5.4.1 MODEL

The model used is a Transformer-based neural network architecture designed for sentiment analysis tasks, particularly for classifying IMDb movie reviews into positive and negative sentiments. It utilizes the Transformer architecture, originally introduced in the context of natural language processing tasks, to process sequential data efficiently.

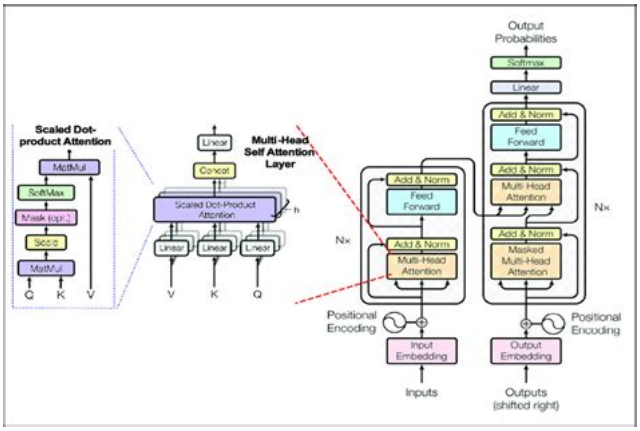


Figure 6: **Transformer Architecture**

The key components of the ImdbNet1 model are as follows:

- **Word Embedding**: The model utilizes pre-trained word embeddings to represent the input text data. These embeddings capture the semantic meaning of words in a continuous vector space, enabling the model to understand the context of the text.
- **Positional Encoding**: Since Transformers do not inherently possess sequential information, positional encoding is applied to the input embeddings to provide information about the position of words in the input sequence. This helps the model differentiate between words based on their positions.
- **Transformer Encoder**: The core of the model is the TransformerEncoder, which consists of multiple TransformerEncoderLayer modules stacked on top of each other. Each TransformerEncoderLayer applies multi-head self-attention mechanisms to capture global dependencies within the input sequence and feed-forward neural networks to process the learned representations.
- **Global Average Pooling**: After encoding the input sequence using the TransformerEncoder, global average pooling is applied to obtain a fixed-size representation of the entire sequence. This aggregated representation captures the most salient information from the input sequence.
- **Fully Connected Layer**: Finally, a fully connected layer is employed to map the aggregated representation to the output classes (positive or negative sentiment).

The hyperparameters used in different instances of the ImdbNet1 model are as follows:

5.4.2 HYPERPARAMETERS

The hyperparameters used in different instances of the ImdbNet1 model are as follows:

- For ImdbNet1 (trans1):
  - Embedding Dimension (emb_dim): 25

  - Number of Attention Heads (num_heads): 5
  - Hidden Dimension (hidden_dim): 64
  - Number of Transformer Encoder Layers (num_layers): 3
- For ImdbNet1 (trans2):
  - Embedding Dimension (emb_dim): 25
  - Number of Attention Heads (num_heads): 5
  - Hidden Dimension (hidden_dim): 64
  - Number of Transformer Encoder Layers (num_layers): 1
- For ImdbNet1 (trans3):
  - Embedding Dimension (emb_dim): 25
  - Number of Attention Heads (num_heads): 5
  - Hidden Dimension (hidden_dim): 128
  - Number of Transformer Encoder Layers (num_layers): 6

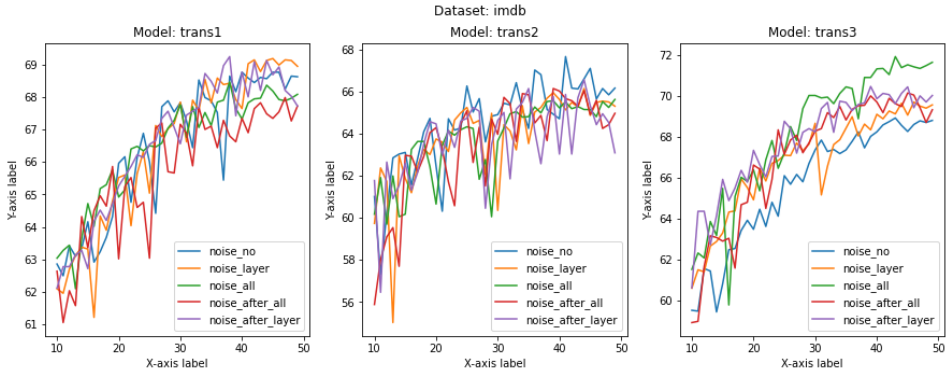


Figure 7: IMDB: Transformers

In summary, the ImdbNet1 model employs the Transformer architecture with varying configurations of attention heads, hidden dimensions, and the number of encoder layers to analyze IMDb movie reviews and predict their sentiment. The model's flexibility and efficiency make it suitable for handling large text datasets and achieving competitive performance in sentiment analysis tasks.

5.4.3 Dataset

The IMDb dataset is selected for its appropriateness in sentiment analysis tasks. It comprises movie reviews annotated with sentiment polarity (positive or negative), making it a suitable choice for training and evaluating text classification models.

5.4.4 Key Observations

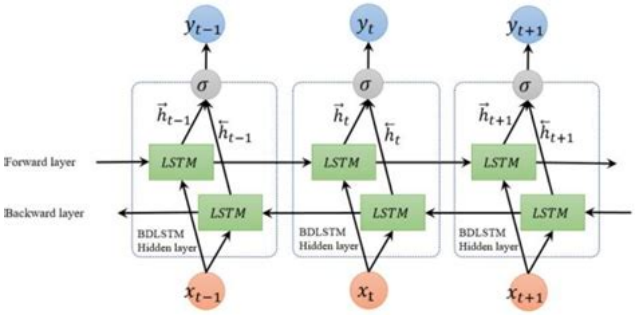
We see surprising results in case of using a transformer. As we increase the model size, adding noise throughout the model actually gives us the best results. This might be related to the fact that transformers operate as single units where they give self attention to the inputs. Adding noise for all of them at the same time might then give better results than adding noise to partial parts of the model.

5.5 Named Entity Recognition (NER)

Named Entity Recognition (NER) is a fundamental natural language processing (NLP) task that involves identifying and classifying named entities in text into predefined categories such as person names, organization names, locations, etc.

5.5.1 MODEL

The BiLSTM model combines Bidirectional Long Short-Term Memory (BiLSTM). The BiLSTM captures contextual information from both past and future words in a sentence.



5.5.2 DATASET

The CoNLL-2003 dataset is commonly used for evaluating NER systems. It consists of English news articles annotated with named entity types such as person names, organization names, locations, and miscellaneous entities. The dataset serves as a benchmark for NER models, allowing researchers to assess their performance on a standardized dataset.

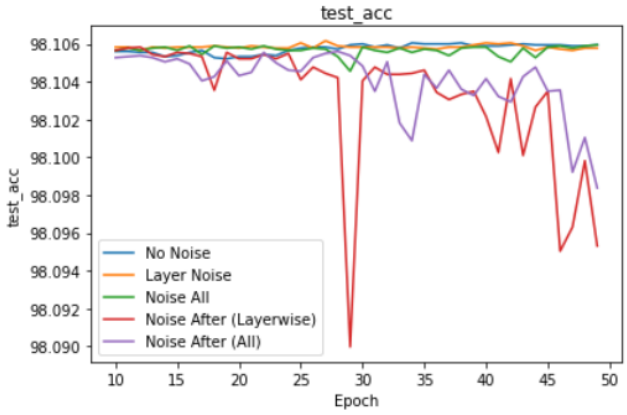


Figure 8: Accuracy using LSTM on NER model

5.5.3 KEY OBSERVATIONS

The layerwise noise gives us the best results. The after noise perform excellent intially but start to drop when more iterations are run.

5.6 TIME SERIES PREDICTION

Time series prediction is a task in which future values of a sequence, typically ordered in time, are forecasted based on historical data. It finds applications in various domains such as finance, weather forecasting, and stock market analysis. One of the commonly used techniques for time series prediction is the Long Short-Term Memory (LSTM) model.

5.6.1 MODEL

The **LSTM model**, a type of recurrent neural network (RNN), is often employed for time series prediction tasks. It is designed to capture long-term dependencies in sequential data by maintaining

a memory cell that can store information over extended time periods. This allows the model to effectively learn patterns and trends in time series data.

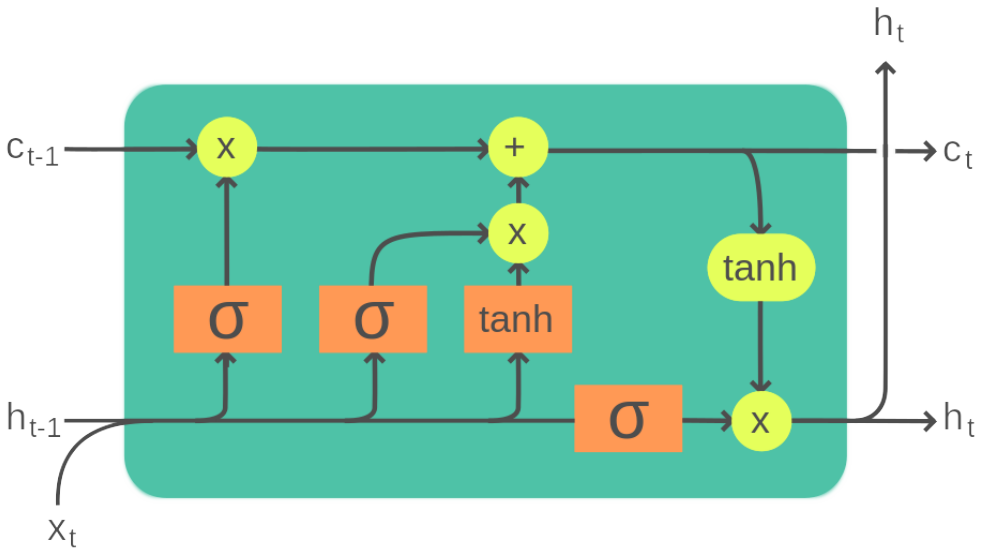


Figure 9: LSTM Cell

In this experiment, we employ Long Short-Term Memory (LSTM) neural networks for time series prediction on the daily minimum temperature dataset. The dataset comprises two columns: the date and the corresponding temperature readings. The rationale behind utilizing LSTM for this task stems from its ability to effectively capture and model temporal dependencies present in sequential data, making it well-suited for time series forecasting tasks.

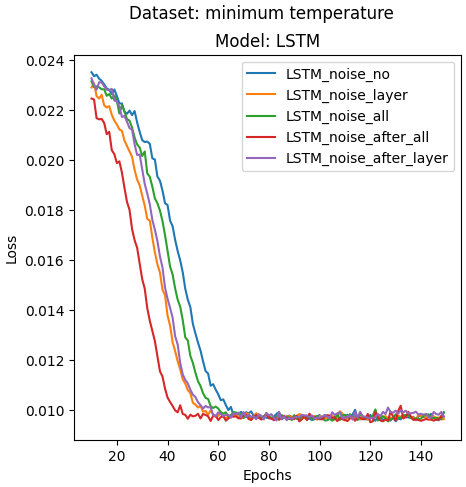
By leveraging LSTM networks, we aim to predict future minimum temperature values based on historical temperature data. The LSTM model learns from the sequential nature of the dataset, capturing temporal patterns and seasonality, which enables it to provide accurate predictions for future time points.

Moreover, we introduce the concept of injecting noise into the gradient descent optimization process during model training. By perturbing the gradients with noise, we aim to explore its impact on the training dynamics and the resulting prediction performance of the LSTM model. This experiment allows us to investigate the potential benefits or drawbacks of incorporating noise into the optimization process, providing insights into its efficacy for enhancing model robustness and generalization capabilities.

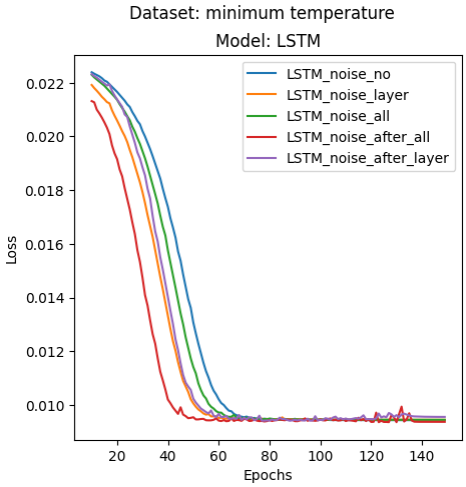
- **Initialization (`__init__`)**:
  - The constructor initializes the LSTM model with input size, hidden size, and number of layers, sets up the LSTM module, and defines two fully connected layers along with the ReLU activation function.
- **Forward Pass (`forward`)**:
  - (`h0` and `c0`) represent the initial hidden and cell states respectively. The input is then passed through LSTM module to get the output which is then passed through fully connected layers (`fc1` and `fc2`), with ReLU activation applied in between.
  - The output of the last fully connected layer (`fc2`) is returned as the final prediction.

5.6.2 Key Observations

So for LSTM model, we observe that over the epoch all the methods achieve almost same accuracy. But the important point is that some methods are going quickly to the optimum

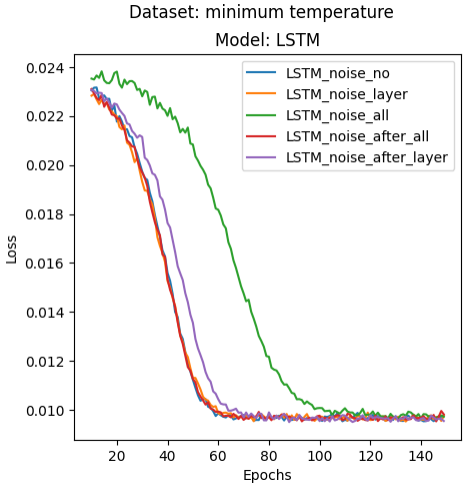


(a) Train Loss

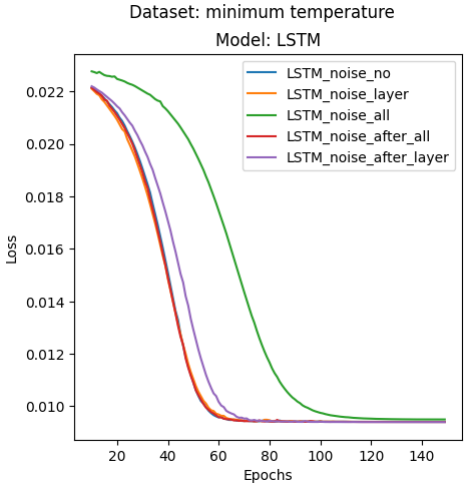


(b) Test Loss

Figure 10: LSTM with $\sigma = 0.003$

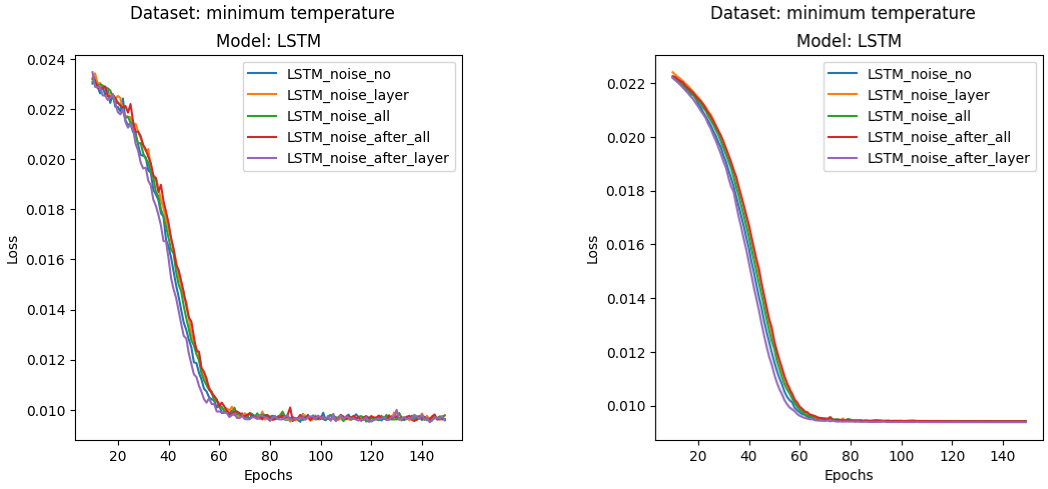


(a) Train Loss



(b) Test Loss

Figure 11: LSTM with $\sigma = 0.00003$



(a) Train Loss

(b) Test Loss

Figure 12: LSTM with $\sigma = 0.000001$

- For $\sigma = 0.003$: We find that the no noise is the slowest of all, incontrast to after-noise on all layers, which is fastest, even faster than layer wise.
- For $\sigma = 0.00003$: We find that if we reduce the noise further then no-noise is the fastest along with others except the before-noise on all layer, which is slowest.
- For $\sigma = 0.000001$: As the noise is very less, hence as expected all the methods perform similarly.

6 Conclusion

In our comprehensive evaluation across various datasets and model architectures, we observed intriguing trends and notable performances. For the FMIST dataset, our findings align closely with those reported in existing literature, demonstrating that both noise addition techniques—layerwise and complete—yield superior results compared to simply adding noise to all layers. Conversely, in the case of CIFAR, we found that after-noise methods yielded suboptimal results, with significant performance degradation. Interestingly, for MNIST, we observed that as the model size increased, layerwise noise addition consistently outperformed other methods, highlighting its efficacy in enhancing model performance. Our experimentation with transformers for IMDB sentiment analysis revealed unexpected outcomes, with noise addition throughout the model yielding the best results, indicating the potential benefits of uniformly applying noise to enhance self-attention mechanisms. In the named entity classification task, layerwise noise emerged as the top-performing method, in handling noise. Finally, in time series analysis using LSTM models, we found that while all methods achieved comparable accuracies over epochs, some methods exhibited faster convergence to optimal solutions, underscoring the importance of noise addition strategies in accelerating model learning. Overall, our findings highlight the nuanced impacts of noise addition techniques across diverse datasets and model architectures, offering valuable insights into optimizing model performance in various domains.

REFERENCES

Antonio Orvieto, Anant Raj, Hans Kersting, and Francis Bach. Explicit regularization in over-parametrized models via noise injection, 2023.

7 APPENDIX

7.1 TEXT CLASSIFICATION

The transformer model which we have used is as follows:

```
class ImdbNet1(nn.Module):
def __init__(self, word_vec, emb_dim, num_heads, hidden_dim,
                                        num_layers, num_classes, device,
                                         dropout=0.1):
    super(ImdbNet1, self).__init__()
    self.word_vec = word_vec.to(device)
    self.emb_dim = emb_dim
    self.positional_encoding = PositionalEncoding(emb_dim, dropout,
                                            device=device)
    encoder_layer = nn.TransformerEncoderLayer(emb_dim, num_heads,
                                            hidden_dim, dropout)
    self.transformer_encoder = nn.TransformerEncoder(encoder_layer,
                                            num_layers)
    self.fc = nn.Linear(emb_dim, num_classes)

def forward(self, x):
    #sent_size = x.size(1)
    embedded = self.word_vec[x]
    embedded = self.positional_encoding(embedded) #.view(-1,
                                            sent_size, self.emb_dim)
    output = self.transformer_encoder(embedded)
    output = output.mean(dim=1)  # Global average pooling
    output = self.fc(output)
    return output
```

7.2 NAMED ENTITY RECOGNITION

The bi-LSTM model we have used is as follows:

```
class BiLSTMNER(nn.Module):
def __init__(self, word_vec, emb_dim, hidden_dim, num_layers,
                                        num_classes, device, dropout=0.1
                                        ):
    super(BiLSTMNER, self).__init__()
    self.word_vec = word_vec.to(device)
    self.emb_dim = emb_dim
    self.hidden_dim = hidden_dim
    self.num_layers = num_layers
    self.num_classes = num_classes
    self.device = device

    self.embedding = nn.Embedding.from_pretrained(self.word_vec)
    self.dropout = nn.Dropout(dropout)
    self.lstm = nn.LSTM(emb_dim, hidden_dim, num_layers=num_layers,
                                            bidirectional=True,
                                            batch_first=True)
    self.fc = nn.Linear(hidden_dim * 2, num_classes)  # Bidirectional
                                             LSTM, so *2

def forward(self, x):
    embedded = self.word_vec[x]
```

```
        embedded = self.dropout(embedded)

        lstm_output, _ = self.lstm(embedded)

        # Concatenate the hidden states of both directions
        lstm_output = torch.cat((lstm_output[:, :, :self.hidden_dim],
                                 lstm_output[:, :, self.hidden_dim:]),
                                                                    dim
                                                                    =2)


        # Apply fully connected layer
        output = self.fc(lstm_output)
        return output.view(-1, self.num_classes, output.size(1))
```

7.3 TIME SERIES PREDICTION

The following is the model code:

```
    class LSTM(nn.Module):

    def __init__( self, inSize , hiddenSize, nLayers ):
        super().__init__()

        self.inSize = inSize
        self.hiddenSize = hiddenSize
        self.nLayers = nLayers

        self.LSTM = nn.LSTM ( input_size = self.inSize,hidden_size = self
                                           .hiddenSize,
                              num_layers = self.nLayers,batch_first = True)

        self.fc1 = nn.Linear( in_features = self.hiddenSize, out_features
                                            = 40)
        self.fc2 = nn.Linear(in_features = 40, out_features =1)
        self.relu = nn.ReLU()

    def forward( self, x):

        h0 = torch.zeros ( self.nLayers, x.size(0), self.hiddenSize,
                                           device=x.device ).float()
        c0 = torch.zeros ( self.nLayers, x.size(0), self.hiddenSize,
                                           device=x.device ).float()

        _ , ( hout , _ ) = self.LSTM( x, ( h0, c0 ) )
        hout = hout.view( -1, self.hiddenSize )
        out = self.fc2( self.relu( self.fc1( hout ) ) )

        return out
```