Noise Injection in Sequential Models

Anonymous submission

Abstract

Sequential Models have gained increasing significance, especially with the dawn of the era of auto-regressive transformer-based language models and their SOTA performance in various domains including text generation, time series prediction and video captioning. In this paper, we investigate the effects of noise injection while training sequential models. Based on the works where Noise Injection has been shown to have regularizing properties in simple neural networks, we consider both the regularizing and the convergence effect that is observed in sequential models e.g. LSTM and Transformer architectures. We use independent layer-wise perturbations as well as all layer perturbations to inject noise during pre-gradient computation and post-gradient computation in our experiments. We have compared the results obtained on training sequential models for classification-based tasks while utilizing different noise injection methodologies. Our experimental results show that adding noise achieves a faster convergence rate in Deep BiLSTM models for NER tasks, and better performance in Narrow Transformer architecture for text classification

Introduction

Perturbations in neural networks have been shown to provide several important benefits. For example, injecting noise during the gradient descent step (Zhu et al. 2019; Nguyen et al. 2019) can help models escape saddle points in non-convex optimization problems and smooth the objective function. Recent studies (Orvieto et al. 2023) demonstrate that noise injection can act as a form of regularization, improving the model's ability to generalize to new data.

Injecting noise into neural network training, especially in adversarial settings, makes models more robust and able to handle disturbances, leading to better performance in real-world scenarios. Recent advances in Natural Language Processing (NLP) have been driven by sequential models like Transformers, which have greatly improved tasks such as text generation and machine translation. Given their wide use and importance, it's crucial to ensure these models are robust.

However, due to the high complexity of pre-trained language models, they often struggle with generalization. Recent work (Hua et al. 2023) has introduced the Layerwise Noise Stability Regularization (LNSR) framework, which improves model robustness by adding noise to inputs in the representation space and regularizing outputs across layers.

Given the critical role of sequential models in modern Natural Language Processing tasks, our paper focuses on enhancing their performance through noise injection techniques. This work explores the application of noise injection to both Transformer and LSTM architectures, aiming to demonstrate its effectiveness in producing more reliable models and better training procedures.

We make the following contributions,

- Incorporated Gaussian noise into the gradient descent algorithm applied to a transformer-based neural network architecture. Our findings indicate that for models with relatively higher dimensions and an increased number of layers, the injection of noise leads to improved accuracy.
- Introduced noise after the gradient descent step in Long Short-Term Memory (LSTM) networks. This approach resulted in a faster convergence rate towards the minima, achieving the same level of accuracy as the traditional gradient descent method without noise.

Related Work

Injecting noise into the input (Bishop 1995), has been shown to have a similar effect to training on clean input with an added regularization term, effectively constraining the solution space. In previous works (Jin et al. 2017; Staib et al. 2020; Reddi et al. 2017), noise injection 'after the gradient step' has been used to escape saddle points efficiently in non-convex objectives.

In the previous work (Orvieto et al. 2023), it has been demonstrated that noise injection during the optimization step has led to explicit regularization. They have theoretically proved for linear neural networks that the expected loss function in case of added noise is equivalent to a vanilla loss with a regularizing term. As the models grew in complexity and size they infered a variance explosion when applying noise injection to overparametrized models. Their 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, they achieved explicit regularization without encountering variance explosion, even in large-scale deep learning models. However, in their work, there was no analysis based on noise injection after the gradient step and was limited to linear networks with experiments on some CNNs as well.

In the paper (Hua et al. 2023), the authors tackle the issues of overfitting and poor generalization when fine-tuning pre-trained language models. They introduce the Layerwise Noise Stability Regularization (LNSR) framework, which improves model stability by adding noise to inputs and regularizing outputs across layers. They use Standard Gaussian and In-manifold noise types. Their results show that LNSR is better at maintaining stability and generalization than several other methods. However, they note that the effectiveness of noise stability might vary between different types of neural networks, like transformers and convolutional networks.

Methodology

Gradient Descent

Gradient Descent is the cornerstone of deep learning and modern AI. It involves first defining a loss function L that denotes how far away we are from getting the ideal result and then optimizing our model weights in such a way that this L reaches its minimum value which is normally 0. When we reach this point, our model performance is extremely close to the perfect solution that can exist.

Gradient Descent is given by

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla f(\theta^{(t)})$$

where

- $\theta^{(t)}$: The vector of parameters at iteration t. These parameters are what the algorithm is optimizing.
- α: The learning rate, a small positive scalar that determines the step size in the direction of the negative gradient. It controls how quickly or slowly the algorithm converges.
- $\nabla f(\theta^{(t)})$: The gradient of the objective function $f(\theta)$ evaluated at $\theta^{(t)}$. This gradient points in the direction of the steepest increase of the function.
- $\theta^{(t+1)}$: The updated parameter vector after taking a step in the direction of the negative gradient.

This happens for multiple iterations and with each iteration, we get closer and closer to the minima of the loss function in general.

One of the most useful modifications to the gradient descent method is to add noise in the update step. There are different ways to do noise injection and in this paper, we discuss two of them - Before Gradient and After Gradient noise injection

Noise Injected Before Gradient Descent: 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 is known to generalize better, thus enhancing the optimization process. The mathematical statement behind this is

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla f(\theta^{(t)} + \epsilon)$$

Noise Injected After Gradient Descent: 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.

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla f(\theta^{(t)}) + \epsilon$$

Complete and Layerwise Pertubation Complete Perturbation is when we inject noise into all the weights present in the model during each iteration. However, this has led to variance explosion especially as neural network width grows to infinity. Another technique as introduced in (Orvieto et al. 2023) is to make the noise perturbation on a random subset of the weights for each iteration. This has been seen to reduce gradient exploding for wide deep networks. This is called Layerwise Perturbation.

Algorithm 1: After and Before GD Noise Injection

```
Require: noise_type, σ
 1: Divide parameters in multiple bins for each layer and store the number of bins as n
 2: for batch in train_loader do
 3:     if noise_type is not None then
 4:         Save model parameters.
 5:         for Each parameter in model do
 6:             if noise_type is All_AfterGD or All_BeforeGD then
 7:                 Add Gaussian noise scaled by $\frac{\sigma}{\sqrt{n}}$ to the parameter.
 8:             else if noise_type is Layer_AfterGD or Layer_BeforeGD then
 9:                 layer is the randomly selected
10:                 Add Gaussian noise scaled by $\sigma$ to the parameters of layer.
11:             end if
12:         end for
13:     end if
14:     Calculate the Gradient and store it.
15:     if noise_type is All_BeforeGD or Layer_BeforeGD then
16:         Restore the original model parameters from the saved copy.
17:     end if
18:     Update model parameters using the previously.
19: end for
```

BiLSTM

A BiLSTM (Huang, Xu, and Yu 2015) is an advanced form of the LSTM (Long Short-Term Memory) network, designed to capture both past and future context in a sequence of data. This is particularly useful in tasks like natural language processing, where understanding the context from both directions is crucial for understanding the model.

Forward LSTM Given an input sequence $\{x_1, x_2, \ldots, x_T\}$, the forward LSTM processes the sequence from $t = 1$ to $t = T$. The hidden states $\overrightarrow{h_t}$ are calculated as follows:

$$\overrightarrow{h_t} = \text{LSTM}(x_t, \overrightarrow{h_{t-1}}, \overrightarrow{c_{t-1}})$$

where $\overrightarrow{h_{t-1}}$ is the hidden state, and $\overrightarrow{c_{t-1}}$ is the cell state at the previous time step. The LSTM cell involves several internal operations:

$$\begin{aligned}
\text{Input gate:} \quad & i_t = \sigma(W_i \cdot [x_t, \overrightarrow{h_{t-1}}] + b_i) \\
\text{Forget gate:} \quad & f_t = \sigma(W_f \cdot [x_t, \overrightarrow{h_{t-1}}] + b_f) \\
\text{Output gate:} \quad & o_t = \sigma(W_o \cdot [x_t, \overrightarrow{h_{t-1}}] + b_o) \\
\text{Cell state update:} \quad & \overrightarrow{c_t} = f_t \cdot \overrightarrow{c_{t-1}} + \, i_t \cdot \tanh(\\
& W_c \cdot [x_t, \overrightarrow{h_{t-1}}] + b_c) \\
\text{Hidden state update:} \quad & \overrightarrow{h_t} = o_t \cdot \tanh(\overrightarrow{c_t})
\end{aligned}$$

Backward LSTM Simultaneously, the backward LSTM processes the input sequence in reverse, from $t = T$ to $t = 1$, generating hidden states $\overleftarrow{h_t}$:

$$\overleftarrow{h_t} = \text{LSTM}(x_t, \overleftarrow{h_{t+1}}, \overleftarrow{c_{t+1}})$$

BiLSTM Output The output of the BiLSTM at each time step t is the concatenation of the forward and backward hidden states:

$$h_t = [\overrightarrow{h_t}; \overleftarrow{h_t}]$$

This concatenated hidden state h_t incorporates information from both past and future contexts.

Architecture We have utilized a BiLSTM model followed by a linear layer for our classification tasks. The model works by taking in a sentence and then converting it to word vectors using the Glove Twitter word2vec pre-trained weights. These vectors are then passed to the BiLSTM and the output from the BiLSMT, containing the concatenation of the forward and backward pass, is then passed through a Linear Layer that will classify it into the k classes available.

Transformer Architecture

The Transformer (Vaswani et al. 2023) is a neural network architecture designed for sequence-to-sequence tasks without relying on recurrent connections. It uses self-attention mechanisms to process sequences in parallel, making it highly efficient and powerful for tasks like machine translation, text generation, and more.

The Transformer consists of an **encoder** and a **decoder**, each composed of multiple layers. Both the encoder and decoder use self-attention mechanisms and feed-forward neural networks.

Encoder The encoder transforms an input sequence into a continuous representation. It consists of multiple identical layers, each containing two main components:

1. Multi-Head Self-Attention Mechanism For an input sequence $X = [x_1, x_2, \ldots, x_T]$, the self-attention mechanism computes attention scores by projecting the input into queries Q, keys K, and values V:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

The scaled dot-product attention for each head is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where d_k is the dimensionality of the keys. Multiple attention heads are used to capture different aspects of the input.

The outputs of all heads are concatenated and linearly transformed:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

2. Position-wise Feed-Forward Network (FFN) After the multi-head attention, a feed-forward network is applied to each position separately:

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2$$

3. Residual Connections and Layer Normalization Both the self-attention output and the feed-forward output are followed by residual connections and layer normalization:

$$\text{LayerNorm}(x + \text{SubLayer}(x))$$

Decoder The decoder generates the output sequence, one token at a time, by attending to both the encoder's output and the previously generated tokens. The decoder also consists of multiple identical layers, with an additional attention mechanism:

1. Masked Multi-Head Self-Attention Similar to the encoder's self-attention, but with masking to prevent attending to future tokens in the output sequence:

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T + M}{\sqrt{d_k}}\right)V$$

where M is a mask that sets the scores of future tokens to $-\infty$.

2. Encoder-Decoder Attention The decoder also performs attention over the encoder's output:

$$\text{Attention}(Q, K', V') = \text{softmax}\left(\frac{QK'^T}{\sqrt{d_k}}\right)V'$$

where K' and V' are the encoder's outputs.

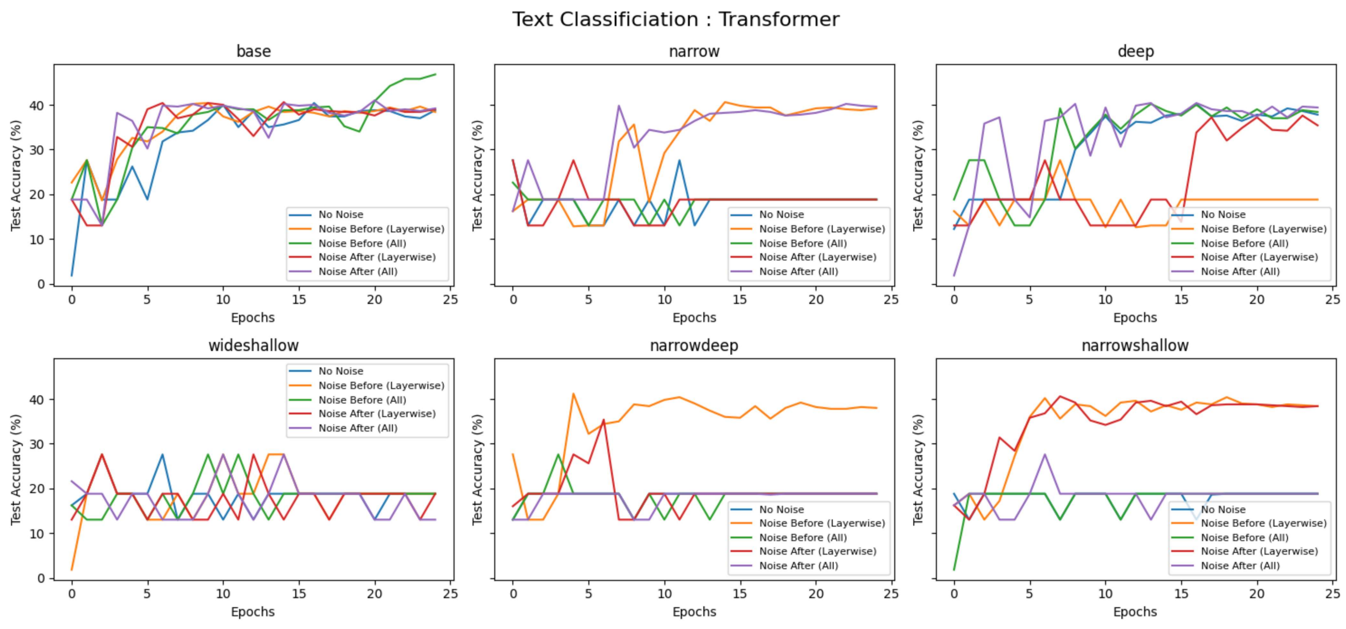

Figure 1: Test accuracies of a transformer model for text classification on the TREC dataset, evaluated using both pre-gradient and post-gradient algorithms. The results include comparisons between layerwise perturbations and all-perturbations strategies.

3. Position-wise Feed-Forward Network (FFN) Same as in the encoder, applied to the output of the attention layers.

4. Residual Connections and Layer Normalization Again, residual connections and layer normalization are applied after each sub-layer.

Positional Encoding Since the Transformer has no inherent notion of sequence order, positional encodings are added to the input embeddings to inject information about the position of each token in the sequence. The positional encoding is defined as:

$$\text{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$
$$\text{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

Complete Architecture

Given an input sequence X and output sequence Y, the Transformer computes:

$$\text{Output} = \text{Decoder}(\text{Encoder}(X), Y)$$

where the encoder transforms the input sequence into a sequence of continuous representations, and the decoder uses these representations along with the previously generated tokens to produce the output sequence.

The architecture used in this paper consists of again utilizing the Glove Twitter word vectors to vectorize the sentence. This is then sent to the Transformer Architecture and its output is passed through 3 linear layers where the number of hidden layers is halved after every layer with the last layer classifying into k classes.

Experiments

In this section, we will discuss in detail the experiment conducted by our models on sequence classification tasks.

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

We use a widely used benchmark collection for evaluating text classification called TREC (Text REtrieval Conference) dataset. The dataset contains 5500 labelled questions in the training set and another 500 for the test set. Each question in the dataset is labelled with one of the 6 different classes such as description, entity, abbreviation, etc. The average length of each sentence is 10, with a vocabulary size of 8700.

We use the transformer architecture described earlier to train on this dataset because of its ability to capture long-range dependencies in text and understand contextual relationships through its self-attention mechanism. We conduct experiments with different hyperparameters for the transformer architecture, primarily by adjusting the model's complexity through widening/narrowing and deepening/shallowing it. Table 1 contains the details of the different hyperparameters used to train the model. In all of the transformer models, the input dimension is 25 and the number of classes is 6, with a dropout rate of 0.1.

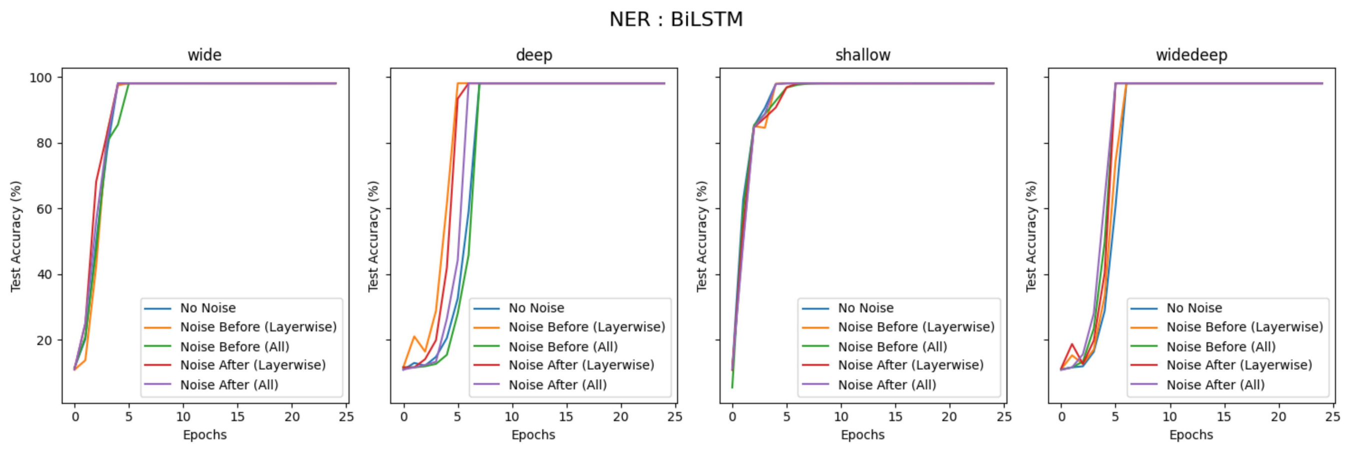

Figure 2: Test accuracies of a BiSTM model for NER task on the CONLL 2003 dataset, evaluated using both pre-gradient and post-gradient algorithms. The results also include comparisons between layerwise perturbations and all-perturbations strategies.

Name	Dim model	Num heads	Num Encoder Layers	Num Decoder Layers	Dim Feedforward
base	50	10	4	4	100
narrow	10	2	4	4	20
deep	50	10	8	8	100
wideshallow	400	80	2	2	800
narrowdeep	10	2	8	8	20
narrowshallow	10	2	2	2	20

Table 1: Transformer model details for text classification.

We observe from Figure 1 that in cases of narrow Transformer architectures, where the dimension of the model is relatively smaller (10 in our case), the accuracy is significantly higher for 'Before Noise (layerwise) method', showing an increase of about 20%.

BiLSTM Model: Named Entity Recognition (NER)

NER is a task which aims to extract information and classify named entities like people, organization, location etc, in a given text input. It is useful as it allows us to automatically understand and organize unstructured text, ultimately improving information accuracy and better decision making.

Here we use the CONLL 2003 dataset (Conference on Computational Natural Language Learning 2003), a widely used benchmark for Named Entity Recognition tasks. The dataset contains text annotated with four types of named entities: Person (PER), Organization (ORG), Location (LOC), and Miscellaneous (MISC). Each word in the text is also labelled with IOB tagging (Inside, Outside, Beginning), thus creating 9 total classes.

We use the BiLSTM architecture described before to train on this CONLL 2003 dataset for the NER task. It catches the context from both the direction of the word, which is crucial for NER tasks as the type of entity often depends on the surrounding words. It is also effective at handling long-term dependencies in sequential data and can accommodate variable-sized inputs. Again we conduct experiments with different-sized models, where Table 2 contains the details of the different hyperparameters used to train the model. In all of the BiLSTM models, the input dimension is 25 and the number of classes is 9.

Name	Dim model	Num Encoder Layers
wide	400	4
deep	50	8
shallow	50	2
widedeep	400	8

Table 2: Named Entity Recognition model configurations.

We observe from Figure 2 that in all cases, the model performed well, achieving nearly the same accuracy whether noise was injected during training or not. However, we noticed that for deeper models, the rate of convergence is faster when noise is introduced during training.

Conclusion

In this paper, we demonstrated the effectiveness of injecting Gaussian perturbations into the gradient descent algorithm to enhance model performance. Our investigation focused on the transformer architecture applied to a text classification task, where we observed that narrower models became more generalizable and achieved higher accuracy with noise injection compared to without noise. Additionally, we analyzed a BiLSTM model trained on the CONLL dataset for a Named Entity Recognition (NER) task. For deeper models, while accuracy remained consistent, we found that the convergence rate significantly improved when noise was introduced.

References

Bishop, C. M. 1995. Training with noise is equivalent to Tikhonov regularization. *Neural Computation*, 7(1): 108–116.

Hua, H.; Li, X.; Dou, D.; Xu, C.-Z.; and Luo, J. 2023. Improving Pretrained Language Model Fine-Tuning With Noise Stability Regularization. *IEEE Transactions on Neural Networks and Learning Systems*, 1–15.

Huang, Z.; Xu, W.; and Yu, K. 2015. Bidirectional LSTM-CRF Models for Sequence Tagging. arXiv:1508.01991.

Jin, C.; Ge, R.; Netrapalli, P.; Kakade, S. M.; and Jordan, M. I. 2017. How to Escape Saddle Points Efficiently. arXiv:1703.00887.

Nguyen, T. H.; Şimşekli, U.; Gürbüzbalaban, M.; and Richard, G. 2019. First Exit Time Analysis of Stochastic Gradient Descent Under Heavy-Tailed Gradient Noise. arXiv:1906.09069.

Orvieto, A.; Raj, A.; Kersting, H.; and Bach, F. 2023. Explicit Regularization in Overparametrized Models via Noise Injection. arXiv:2206.04613.

Reddi, S. J.; Zaheer, M.; Sra, S.; Poczos, B.; Bach, F.; Salakhutdinov, R.; and Smola, A. J. 2017. A Generic Approach for Escaping Saddle points. arXiv:1709.01434.

Staib, M.; Reddi, S. J.; Kale, S.; Kumar, S.; and Sra, S. 2020. Escaping Saddle Points with Adaptive Gradient Methods. arXiv:1901.09149.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; and Polosukhin, I. 2023. Attention Is All You Need. arXiv:1706.03762.

Zhu, Z.; Wu, J.; Yu, B.; Wu, L.; and Ma, J. 2019. The Anisotropic Noise in Stochastic Gradient Descent: Its Behavior of Escaping from Sharp Minima and Regularization Effects. arXiv:1803.00195.