Enhancing NanoGPT via Squentropy Loss and Hyperparameter Tuning

Akshat Muir akmuir@ucsd.edu

Sujay Talanki stalanki@ucsd.edu

Rehan Ali rmali@ucsd.edu

Sujen Kancherla skancherla@ucsd.edu

Misha Belkin mbelkin@ucsd.edu

Yian Ma yianma@ucsd.edu

Abstract

In the field of natural language processing, there is a focus to make large language models (LLM's) compact while maintaining efficiency; NanoGPT is a promising model to address resource constraints without compromising performance. However, there are questions surrounding the model: In what ways can the model's performance be improved? This paper introduces methods in an attempt to improve performance: (1) optimize the *squentropy* (Like Hui 2023) loss function and (2) choose optimal hyperparameters. Traditionally, most language models attempt to minimize the cross entropy loss function when predicting the next token in the sequence. Squentropy (2) is a hybrid loss function combines the formulas of cross entropy loss (1) and mean squared error loss, and has shown promising results in other applications. The implementation is difficult because the output tensors need to be manipulated and reshaped to be used for the correct squentropy calculation. The implementation is mathematically intensive, relying on vector algebra and probability theory. The mathematical steps were translated into python code. After implementing the changes and performing hyperparameter tuning (will be discussed in future section) and testing the model on various datasets, it seems as if squentropy loss could work almost as effectively as cross entropy loss. The new model produced effective results in terms of perplexity (3) and loss, but did not beat the baseline cross entropy model's performance (which seems to be the most effective).

Code: `https://github.com/BigSuj/NanoGPT-Loss-Stop-Analysis/tree/main`

Website: `https://akshatm1011.github.io/Optimizing-NanoGPT/`

Poster: `https://publuu.com/flip-book/432465/976540`

1 Introduction

In the context of LLM's, there has been a growing interest in improving the performance of compact models such as NanoGPT. These models not only aim to generate coherent text but also strive to optimize resource utilization in text generation tasks. We are interested in optimizing NanoGPT's performance; we will particularly focus on the loss function that the model attempts to minimize and hyperparameters that can be tuned. In machine learning, a model makes a prediction by choosing the input that minimizes a loss function. LLM's traditionally use cross entropy Loss function, primarily because it is well-suited for tasks that generate probabilistic predictions (and classification tasks in general). The MSE loss function is typically utilized for regression tasks (It uses the residuals- the error between the predicted value and the actual value.). However, there is a way to utilize the MSE loss function for our application. It involves predicting the next token in the sequence (choosing the token with the maximum probability of occurring) and comparing this token to the actual token. These residuals will be computed through vector algebra, and inputted into the loss function to compute a final metric. The goal is to implement this mathematical transformation into code, and evaluate the result on our dataset to see if the model performs better. This research aims to contribute to both the advancement of NanoGPT and other large language models in the field of natural language processing.

1.1 Discussion of Prior Work

There has been some work building NanoGPT models. Specifically, Andrej Kaparthy has built an effective NanoGPT model (Karpathy 2023) that can generate output text effectively based on input text data. The model uses tokenization, transformers, and many deep learning techniques to build a smaller scale yet effective GPT model. However, Kaparthy stated in the README file that the current set of model parameters are poor and could use tuning. Thus, this project focuses on optimizing NanoGPT by choosing the right loss function and tuning its hyperparameters. Hyperparameters that can be adjusted include the dropout rate, step size, number of layers in the network, the maximum number of training iterations, the early stopping condition, and many others. We will focus on the learning rate, dropout rate, and number of layers! Kaparthy was able to train the NanoGPT model using a singular GPU (although there is an option to use CPU's for model training, if one did not have access to a GPU). The model seems to run in a few minutes on smaller datasets, but takes many days to run on bigger datasets (datasets with thousands of rows containing many "tokens"). To summarize, the NanoGPT model infrastructure has been set up effectively, but can be improved with a few variations regarding the loss function and hyperparameters utilized.

1.2 Dataset Description

There are multiple datasets that we would like to explore, but we will be using TinyStories (Eldan 2023). This dataset has 2.1M rows; each row consists of a short, structured narrative. This dataset differs from most datasets, as there is technically only one independent variable (the input paragraph). The data does not need to be necessarily cleaned (some grammar or punctuation might need to be handled, but nothing that requires too much attention). During the preparation process, the code utilizes tokenizers, transformers, and deep learning techniques to generate output text based on the input. The choice of dataset is actually not too important for this task, as there are many datasets of raw text data that can be used for NanoGPT training. However, the TinyStories dataset works particularly well because the dataset is large and contains many tokens that can be used in training. As far as limitations go, this dataset takes approximately 1 day to train when using a singular node with a singular GPU. (Runtime will be significantly reduced if we use Distributed Parallel Computing methods.) Even for our small scale NanoGPT infrastructure, running the model on this dataset requires much processing time.

2 Methods

2.1 Overview

In this study, we aimed to assess the performance of the GPT-2 model, consisting of 127 million parameters, by training it on the Tiny Stories dataset. The primary focus was to measure the model's performance post-training using a dedicated script. The training was conducted using PyTorch, and Weights & Biases (wandb) was employed for visualizing the loss over iterations. This section delineates the methodology in detail, ensuring clarity and reproducibility while maintaining conciseness to keep the reader focused on the primary narrative.

2.2 Model Configuration

The GPT-2 model used in this study is a smaller variant with 127 million parameters. This size was chosen to balance computational demand with the ability to capture complex patterns in the data. The model's architecture, pre-defined in the PyTorch library, was utilized without modification to maintain consistency with standard GPT-2 structures.

2.3 Training Procedure

Training was conducted on a GPU-enabled environment to manage computational requirements effectively. The model was trained iteratively, with checkpoints at regular intervals to monitor progress and prevent data loss. The learning rate, batch size, and other hyperparameters were set following preliminary tests to determine optimal values for this specific dataset and model configuration. A singular node and GPU were utilized to train each model for a period of 14 days.

2.4 Dataset and Preprocessing

The Tiny Stories dataset, comprising short, structured narratives, was selected for its diversity and size, suitable for training a model of this scale. Data preprocessing involved tokenization and formatting to align with the input requirements of the GPT-2 model. Care was taken to ensure data integrity and fair representation throughout the preprocessing steps.

2.5 Current Model: Cross Entropy Loss

The current NanoGPT model optimizes the cross entropy loss function, which is defined below:

Consider the following notation:

- Let $D = \{(x_1, y_1), ..., (x_n, y_n)\}$ denote the dataset sampled from a joint distribution $D(X, Y)$
- For each sample $i, x_i \in X$ is the input and $y_{i,j} \in Y = \{0, 1\}$ denotes if label j is the correct classification for observation i
- Let $f(x_i) \in \mathbb{R}$ denote the logits (output of last linear layer) of a neural network of input x_i, with components $f_j(x_i)$, $j = \{1, 2, ..., C\}$
- Let $p_{i,j} = \frac{e^{f_j(x_i)}}{\sum_{j=1}^{C} f_j(x_i)^2}$ denote the predicted probability of x_i to be in class j.

$$L_{CE}(x, y_i) = -\sum_{c=1}^{C} y_{i,j} \log(p_{i,j}) \tag{1}$$

Most classification tasks optimize the cross entropy loss function, but we attempt to implement a hybrid *squentropy* loss function, described in the next section.

2.6 Implementing Squentropy Loss

The squentropy loss function is a hybrid loss function that combines aspects of cross entropy loss and mean squared error. The squared loss portion of this function acts similar to a *regularization* term in traditional machine learning applications (can be used to reduce overfitting)!

Consider the following notation:

- Let $D = \{(x_1, y_1), ..., (x_n, y_n)\}$ denote the dataset sampled from a joint distribution $D(X, Y)$
- For each sample $i, x_i \in X$ is the input and $y_i \in Y = \{1, 2, ..., C\}$ is the true class label
- The one-hot encoding label used for training is $e_{y_i} = [0, ..., 1, ..., 0] \in \mathbb{R}^C$
- Let $f(x_i) \in \mathbb{R}$ denote the logits (output of last linear layer) of a neural network of input x_i, with components $f_j(x_i)$, $j = \{1, 2, ..., C\}$
- Let $p_{i,j} = \frac{e^{f_j(x_i)}}{\sum_{j=1}^{C} f_j(x_i)^2}$ denote the predicted probability of x_i to be in class j.

Then the squentropy loss function on a single sample x_i is defined as follows:

$$L_{squen}(x_i, y_i) = -\log p_{i,y_i}(x_i) + \frac{1}{C-1} \sum_{j=1, j \neq y_i}^{C} f_j(x_i)^2 \qquad (2)$$

The first term $-\log p_{i,y_i}(x_i)$ is simply cross-entropy loss. The second term is the square loss averaged over the incorrect ($j \neq y_i$) classes. The code is shown in A.2

2.7 Hyperparameter Tuning

LLM performance can be improved by choosing the optimal hyperparameters in the training process (via hyperparameter tuning). For our purposes, we chose to change the learning rate, number of layers in the model, and the dropout rate. Using our hyperparameter tuning script (A.3), we implemented a grid search: exhaustively search all possible combinations of hyperparameter values within our defined search space (refer to A.3 to see search space). We chose the model with the hyperparameters that resulted in the lowest perplexity metric. The most optimal hyperparameters found for squentropy are written below.

- Learning rate - 0.00006
- Number of Layers - 16
- Dropout rate - 0.1

2.8 Perplexity Measurement

Upon completion of the training, *perplexity* was measured using a separate script (shown in A.1). Perplexity measures how well a language model predicts or understands a given set of data, typically a sequence of words or tokens (Madiraju 2022). The lower the perplexity, the better the model is at making accurate predictions. It quantifies how surprised or ”perplexed” the model would be on average when seeing a new word. The script calculated the perplexity for each story in the dataset, providing a comprehensive view of the model's performance.

The formula is shown below:

Consider the following notation:

- N is the number of tokens in the test set
- w_i represents the i-th word in the test set
- $P(w_i|w_1, w_2, ..., w_{i-1})$ is the probability assigned by the language model to the i-th token given the previous words

Then, the *perplexity* is defined as:

$$P(w) = exp(\frac{-1}{N}\sum_{i=1}^{N}\log P(w_i|w_1, w_2, ..., w_{i-1})) \tag{3}$$

2.9 Visualization and Monitoring

Throughout the training process, Weights & Biases (wandb) was used for real-time monitoring and visualization. This tool provided insights into the model's learning progression by graphically representing the loss over iterations. Wandb's interactive dashboards facilitated a deeper understanding of the training dynamics and helped identify any anomalies or areas for optimization.

2.10 Ethical Considerations and Limitations

Ethical considerations, particularly regarding data privacy and model misuse, were addressed. The Tiny Stories dataset, being publicly available and non-sensitive, mitigated privacy concerns. The potential for the model to generate inappropriate content was acknowledged, and limitations in its applicability were discussed.

3 Results

The graphs below show each model's loss for each iteration during training. The two models were trained on a training set and evaluated on a test set:

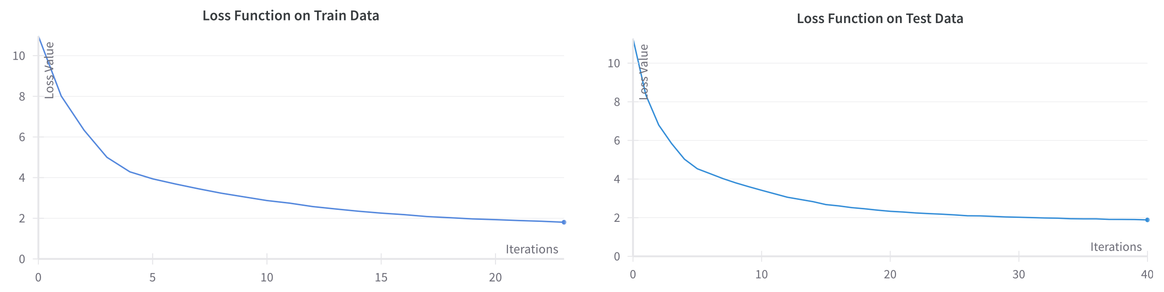

Figure 1: Losses for Training and Test set using Cross Entropy Loss Function

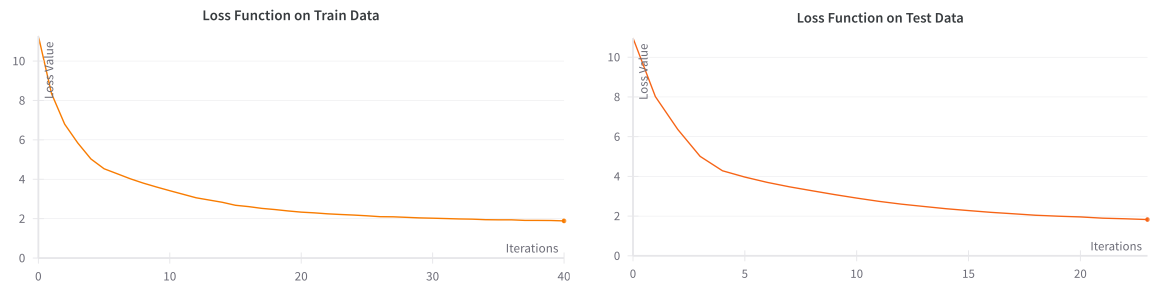

Figure 2: Losses for Training and Test set using Squentropy Loss Function

The loss and perplexity metrics are shown below:

Table 1: Metrics for Each Loss Function

Method	Cross Entropy	Squentropy (Optimal Hyperparameters)
Train Loss	1.805	1.885
Test Loss	1.826	1.899
Test Perplexity	3.8	5.2

A sample output for each model is shown below:

3.1 Cross Entropy

Once upon a time, there was a little girl named Lily. She was so tired because she didn't want to play. But before she started to feel dizzy, she started to feel a little better.

Suddenly, she heard a noise outside. She looked up and saw a little mouse named Tom. He said, "Hi Lily, what are you doing?"

Lily replied, "I'm just playing!"

Tom looked up and said, "I'm teasing you. Can I try again?"

Lily was so happy to hear this and said, "Sure, you can try another game next time." Tom was so happy! He smiled and said, "Thanks for letting me play."

Lily and Tom continued to play together until the sun started to go down. Then they went back to their homes and Lily cried. "Thank you for the game, Tom!" The end.

3.2 Squentropy

Once upon a time, there was a big dog named Max. Max was very grumpy because he did not like to play with his friends. One day, Max's friends wanted to play a game of catch with Max's friends. They all ran to the pitch and started to play.

Max was very good at catch the ball very far. He didn't know that his friends would try to catch it and get it back. His friends were happy to hear him and wanted to play too. Max felt happy that his friends were happy too.

As they played, Max accidentally hit a big tree. His friends tried to help him but Max was still grumpy. Finally, his friends got cross and Max was very happy. His friends were proud of him for being good and telling the truth. Max learned that being grumpy is not a good thing to do. From then on, Max and his friends played with other dogs and had a lot of fun together. The end.

4 Discussion

During this process, we chose to depart from cross entropy loss and opt into squentropy loss in pursuit of better performance. When evaluating performance, perplexity is commonly used instead in addition to validation loss because perplexity indicates the model's effectiveness at predicting the next word in the sequence. Our tuned squentropy model performed decently well, achieving a perplexity of 5.2 and MSE loss of 1.899. However, the cross entropy model outperformed these results, achieving a perplexity (3.8) and validation loss of 1.83. Although the squentropy loss performed decently well (even by looking at the output, the sentences are very readable and look human generated), it is apparent that the cross entropy loss performs the best.

5 Conclusion

The current state of machine learning and natural language processing is heavily focused on LLM's. ChatGPT has revolutionized the industry, and has directed the focus of AI towards LLM research and optimization. These recent events have inspired us to improve upon NanoGPT's performance. Current research suggest the cross entropy loss function performs optimally, as it's suited for classification tasks and probabilistic predictions. Nonetheless, we attempted to tune a squentropy loss function. Although our attempt to optimize NanoGPT was futile, we did learn that the squentropy loss with tuned hyperparameters could perform almost as effectively. This is surprising because squentropy has a squared loss component. Squared loss is used typically for regression tasks, so using the squentropy loss function seemed intuitive at first. To summarize, cross entropy loss seems like the best loss function for this task.

References

Eldan, Ronen. 2023. “TinyStories.” *Hugging Face*. [Link]
Karparthy, Andrej. 2023. “nanoGPT.” *GitHub*. [Link]
Like Hui, Stephen Wright, Mikhail Belkin. 2023. “Cut your Losses with Squentropy.” *Cornell University*. [Link]
Madiraju, Priyanka. 2022. “Perplexity of Language Models.” *Medium*. [Link]

Appendices

A.1 Code to Compute Model Perplexity

```python
def calculate_perplexity(model, device, num_batches):
    model.eval()
    total_loss = 0
    total_count = 0

    with torch.no_grad():
        for _ in range(num_batches):
            inputs, targets = get_val_batch()
            with ctx:
                logits, loss = model(inputs, targets)
            log_probs = F.log_softmax(logits, dim=-1)

            # Reshape log_probs and targets for calculating loss
            # Flatten targets to [batch_size * sequence_length]
            targets = targets.view(-1)
            # Flatten log_probs to [batch_size * sequence_length,
                    vocab_size]
            log_probs = log_probs.view(-1, log_probs.size(-1))

            # Calculate the loss
            loss = F.nll_loss(log_probs, targets, reduction='sum')

            # Adds to Total Loss
            total_loss += loss.item()

            # Counts number of target tokens
            total_count += targets.numel()

    average_loss = total_loss / total_count

    # Calculates Perplexity of Model
    perplexity = np.exp(average_loss)
    return perplexity
```

Listing 1: Perplexity Script

A.2 Code to Perform Squentropy Calculation

```python
def mse_loss(targets_expanded, logits, vocab_size):
    squared_error = (targets_expanded - logits)**2
    targets = targets_expanded == 1
    squared_error = torch.where(targets, squared_error * vocab_size, squared_error)
    mse_loss = torch.mean(squared_error)
    return mse_loss

loss = mse_loss_value + cross_entropy_loss
```

Listing 2: Squentropy Code

A.3 Code to Perform Hyperparameter Tuning

```python
import os
from itertools import product

learning_rates = [0.000006,0.0006,0.06]
dropouts = [0.0, 0.1, 0.2]
n_layers = [8, 12, 16]


params = list(product(learning_rates, dropouts, n_layers))
path = os.getcwd().split('/')[:3]
path += ['teams', 'b13', 'group1']
out = os.path.join(*path)

counter = 0
for lr, dropout, n_layer in params:
    command = f'python3 train.py --compile=False --wandb_log=True --out_dir={out} --
    batch_size=4 --max_iters=50 --eval_interval=50 --loss_func="squentropy" --
    learning_rate={lr:.9f} --min_lr={lr/10:.9f} --dropout={dropout} --n_layer={n_layer}
     --ckpt_name=ckpt{counter}.pt'
    #print(command)
    os.system(command)

    counter += 1
```

Listing 3: Hyperparameter Tuning Script