Interim report

As promised before, more advanced models have been and being developed to outperform previous ones and get better loss and generations.

Right now there are five major models developed. And each next model outperforms the previous ones as the complexity of the model gets bigger.

- Bigram
- Trigram
- MLP
- Wavenet
- RNN

Bigram, Trigram Implementation

Bigram, Trigram and Trigram neural network models (the performance of this is identical to count-based one) have been trained as a starter base models.

- The code calculates the probability distribution over the first character of the n-gram by selecting the corresponding row N[0] from the count matrix N.
- The probabilities are converted to floating-point values and stored in the variable p.
- To ensure the probabilities sum to 1, p is divided by the sum of its elements using p / p.sum().

In summary, the count-based n-gram model architecture relies on counting the occurrences of bigrams in a given dataset. The code iterates through a list of words, generates character sequences, and updates the count matrix accordingly. It then calculates the probability distribution over the first character of the bigram based on the counts. This model can be used to estimate the likelihood of generating the next character given the previous character in a sequence.

Neural network version of those n-gram models also performs same as the count-based model just because those models are simple and the parameters that learns the data is considerably low. This implementation includes optimizing one multidimensional (27 by 27 for bigram, 729 by 27 for trigram) weight matrix.

MLP Implementation

```
n_emb = 10 # dimension of the feature vector (embedding)
n_hid = 200 # number of neurons in the hidden layer
g = torch.Generator().manual_seed(10110609)       # seed to to get same result
C = torch.randn((27, n_emb), generator=g)         # feature vector for each char
W1 = torch.randn((n_emb*block_size, n_hid), generator=g)  # hidden layer weights
b1 = torch.randn(n_hid, generator=g)                      # hidden layer biases
W2 = torch.randn((n_hid, 27), generator=g)       # weigths for the output layer
b2 = torch.randn(27, generator=g)                # biases for the output layer
params = [C, W1, b1, W2, b2]
```

The MLP architecture consists of three main components: an embedding layer, a hidden layer, and an output layer.

1. Embedding Layet:

- The embedding layer maps each character to a continuous feature vector representation of dimension `n_emb`.
- In the code, the embedding matrix `C` is of size (27, n_emb), where 27 represents the total number of characters in the vocabulary, and `n_emb` represents the dimension of the feature vector.
- Each row in the embedding matrix `C` corresponds to the feature vector representation of a specific character.

2. Hidden Layer:

- The hidden layer contains `n_hid` neurons and applies the Rectified Linear Unit (ReLU) activation function.
- The input to the hidden layer is obtained by flattening the embedded input sequence and applying matrix multiplication with the weight matrix `W1` of size (n_emb * block_size, n_hid), where `block_size` represents the size of the input block (number of characters in a sequence).
- The biases of the hidden layer are represented by the vector `b1`, which has a size of n_hid.
- The hidden layer applies the ReLU activation function element-wise to the linear combination of inputs and biases.

3. Output Layer:

- The output layer generates logits (unnormalized probabilities) for each character in a 27-class output (assuming a 26-character alphabet plus one additional symbol).
- The input to the output layer is obtained by multiplying the hidden layer's output with the weight matrix `W2` of size (n_hid, 27).
- The biases of the output layer are represented by the vector `b2`, which has a size of 27.

In summary, the MLP architecture takes a sequence of characters as input, applies character embedding, feeds it through a hidden layer with ReLU activation, and generates logits for each character in the output layer. The architecture learns to capture the relationships between characters in the input sequence and predicts the probabilities of the next character.

WaveNet Architecture

Next, I explored the implementation of WaveNet, a deep generative model known for its ability to generate high-quality audio waveforms. While WaveNet is primarily designed for audio tasks, I adapted it for character-level language modeling to leverage its autoregressive nature and capture complex dependencies among characters.

The WaveNet architecture is composed of dilated causal convolutional layers, which allow the model to condition the prediction on past characters. The model generates the probability distribution of the next character given the previous characters. In this adaptation, I considered the discrete nature of characters and modified the output layer to predict the probability distribution across the character vocabulary.

Implementation Details: To implement WaveNet for character-level language modeling, I followed the guidelines and insights from the original WaveNet paper (van den Oord et al., 2016) and related adaptations. The description of the the main components of this implementation below:

Model Architecture: I configured the dilated causal convolutional layers with appropriate dilation factors to capture short and long-range dependencies among characters.

Training Data: I used the cleaned and preprocessed dataset names to train the WaveNet model. Each character was treated as a discrete category, and appropriate preprocessing steps were applied to convert the characters into numerical representations.

Training Procedure: I utilized a variant of the teacher-forcing technique during training, where the true preceding characters were used as input to predict the next character. I employed techniques like backpropagation through time and gradient descent to optimize the model's parameters.

Hyperparameter Tuning: I performed extensive hyperparameter tuning to optimize the performance of the WaveNet model. Key hyperparameters included the number of layers, dilation factors, learning rate, batch size, and regularization techniques.

Evaluation and Results
To evaluate the WaveNet model, I employed standard evaluation metrics such as cross entropy, negative log likelihood. These metrics quantified the model's ability to generate accurate and coherent sequences of characters. I compared the performance of WaveNet with other models, such as bigram, trigram, MLP, and RNN, to assess its effectiveness which showed that this outperforms those simpler models.

Design of the architecture from the high level:

```
# hierarchical network
n_embd = 24 # the dimensionality of the character embedding vectors
n_hidden = 128 # the number of neurons in the hidden layer of the MLP
model = Sequential([
  Embedding(vocab_size, n_embd),
  FlattenConsecutive(2), Linear(n_embd * 2, n_hidden, bias=False), BatchNorm1d(n_hidden), Tanh(),
  FlattenConsecutive(2), Linear(n_hidden*2, n_hidden, bias=False), BatchNorm1d(n_hidden), Tanh(),
  FlattenConsecutive(2), Linear(n_hidden*2, n_hidden, bias=False), BatchNorm1d(n_hidden), Tanh(),
  Linear(n_hidden, vocab_size),
])
```

In summary, this hierarchical WaveNet architecture employs character embeddings, multiple layers of FlattenConsecutive and Linear operations, batch normalization, and Tanh activation functions. It aims to capture both local and global dependencies within the character sequence and generate output probabilities for character-level language modeling.

- Embedding layer: The first layer is an embedding layer, which maps the characters to continuous vector representations (embeddings) of size n_embd. The vocab_size indicates the total number of unique characters in the dataset.
- FlattenConsecutive layer: The next three pairs of layers utilize the FlattenConsecutive and Linear operations in a hierarchical manner. The FlattenConsecutive operation takes consecutive sets of characters, flattens them, and concatenates them. This allows the model to capture both local and global dependencies within the character sequence. The Linear layers perform linear transformations on the concatenated representations.
- BatchNorm1d layer: Batch normalization is applied after each Linear layer to improve training stability and performance. It normalizes the activations within each mini-batch, reducing internal covariate shift.
- Tanh layer: The Tanh activation function is used after each BatchNorm1d layer to introduce non-linearity to the network.
- Linear layer: The final Linear layer maps the hidden representations to the vocabulary size, producing output probabilities for each character in the vocabulary.

RNN Implementation

This implementation consists of two main components: the RNNCell and the RNN model.

1. RNNCell:
   - The RNNCell is responsible for processing input at each time step and updating the hidden state. It takes the current input `xt` and the previous hidden state `hprev` as input and returns the updated hidden state `ht` at the current time step.

- In this implementation, the RNNCell uses a linear layer (`xh_to_h`) to transform the concatenation of the current input and previous hidden state. The transformed output is passed through a Tanh activation function to produce the updated hidden state.

2. RNN Model:
- The RNN model handles the sequential processing of the input sequence and generation of output logits.
- The model first initializes the hidden state (`hprev`) with a learnable parameter (`self.start`).
- The input sequence is embedded using an embedding layer (`self.wte`) that maps integer indices to continuous vector representations.
- The embedded inputs are sequentially fed into the RNN cell (`self.cell`) along with the current hidden state. The RNN cell updates the hidden state at each time step.
- The hidden states obtained at each time step are stored in a list (`hiddens`).
- The hidden states are then passed through a linear layer (`self.lm_head`) to produce output logits representing the predicted probabilities for each token in the vocabulary.
- If target values are provided, the model calculates the cross-entropy loss between the predicted logits and the targets using the `F.cross_entropy` function.
- The logits and loss (if applicable) are returned as output from the model.

Overall, this RNN implementation processes the input sequence one step at a time, updating the hidden state at each step, and generates output logits representing the predicted probabilities for each token in the vocabulary. It allows for both forward pass inference and training with optional loss calculation.

As the model architecture gets more complex I was able to get better loss function and better word sampling from the models. In the previous report I mentioned that the best loss function was 2.363 by the MLP. The result for the Wavenet and the RNN is 2.2562 and 2.1814 respectively. And below are some sampling from the model:
revicore, blyndo, scuarx, autiv, bantist, talense, cooco, webtue