Code Analysis Questions

For most of the questions below, I used information from the blog post by the author of the RNN code, referenced below:

Karpathy, A. (2015, May 21). The Unreasonable Effectiveness of Recurrent Neural Networks. Andrej Karpathy Blog. https://karpathy.github.io/2015/05/21/rnn-effectiveness/

2.1 - Understanding the Model

a) What does each matrix (W_{xh}, W_{hh}, W_{hy}) represent?

Answer: As mentioned by Karpathy (2015), each matrix represents a matrix of learned (updated after each iteration of the training process) weights involved in the RNN.

The W_{xh} matrix has dimensions given by (hidden state size $\times$ number of unique characters). A single character is one-hot encoded, so the number of unique characters is the length of the vector representing a single character read into the model. The W_{xh} matrix is then used in the linear operation $W_{xh}x$ (which is only part of the sum of linear outputs used as input to the activation function).

The W_{hh} matrix is a square matrix with dimensions equal to the hidden state size. Within the forward pass, the hidden state from the previous iteration of the character generation is multiplied by W_{hh} (via the linear operation $W_{hh}h_{t-1}$, where h_{t-1} is the previous iteration's hidden state vector). The output of this operation is added with the output $W_{xh}x$ from the most recently-processed character, along with a bias term, and the tanh activation function is applied to this combination to get the new hidden state vector.

The W_{hy} matrix has dimensions (number of unique characters $\times$ hidden state vector size). It is a matrix of weights multiplied by the current hidden state output in a linear layer with bias ($W_{hy}h_t + b_y$), the output of which is fed into the softmax to predict the next character.

b) Why do we use the tanh activation function in the hidden state update?

Answer: The use of the tanh activation function applied to the sum of the hidden→hidden and input→hidden linear terms (plus the bias) allows for nonlinearities in the model. If no activation function was used, the RNN could only model linear relationships between the hidden state, input character, and output character, limiting its potential to understand more complex relationships and likely reducing its accuracy.

c) How is the hidden state initialized, and why?

Answer: The hidden state is initialized as a zero vector. This is because, when the model begins training, it has no prior context (in the form of having seen previous characters that could provide useful information about the next character). Therefore, specifying a nonzero hidden state could introduce bias into the model (favor certain initial characters). To avoid this, the hidden state is set to a vector of zeros, making the product $W_{hh}h_0$ at the first iteration of the forward pass equal to zero. As a result, the output after the first iteration depends entirely on the linear output $W_{xh}x + b$.

2.2 - Training Process

a) What loss function is used, and why?

Answer: A categorical cross-entropy loss function is used. This is the appropriate loss function for the task of the RNN during the training process, which is correctly classifying the next character in a string. There are a fixed number of unique characters in the training dataset, and the RNN outputs a vector of probabilities that the next character is each of these unique characters (the softmax output). Therefore, the problem is a classification one – the RNN is attempting to identify which of the classes (unique characters) the next character belongs to. Therefore, the categorical cross-entropy loss is most appropriate (the explanation of this – based on the fact that the categorical cross-entropy loss is derived from MLE applied to the softmax output – was noted in HW1) (Prince, 2024).

b) How does the model update its weights?

Once the model computes the gradients via backpropagation, it uses what is referred to in the comments of the code (Karpathy, 2015) as AdaGrad. As implemented, the derivatives of each parameter value are multiplied by the learning rate as usual. However, this value is then divided by *mem*, which is the cumulative sum of the squared changes of the parameter from all previous training iterations (plus a very small constant). Therefore, the parameters in later iterations are changed at a relatively slower rate compared to the parameters in earlier iterations, and the rate of parameter change as a function of iteration is lower when the parameters were changed substantially in past iterations.

c) What is the purpose of gradient clipping (np.clip)?

The gradient clipping process is intended to "mitigate exploding gradients" as noted by Karpathy in the comments of the RNN code (code linked in Karpathy (2015)). Effectively, this means that the process is intended to to avoid gradients that blow up in the model and cause numerical issues. If very large gradients are produced during one iteration of backpropagation, resulting weights would be significantly adjusted. This could potentially move some weights relatively far from an optimal value, resulting in more large gradients and more significant adjustments to weights. This process could repeat, causing larger and larger fluctuations away from the optimum until numerical issues are encountered. With gradient clipping, these issues are mitigated because gradients are not permitted to exceed a certain absolute value. This could limit the speed of model training in some cases, as more iterations may be necessary to get close to optimal weights, but it increases the chance that the model successfully trains in the first place.

2.3 - Text Generation

a) How does the model generate text?

Answer: Note: This text generation process is done in the sampling function, so my answer to this question and part (b) are similar. The model generates the next character by taking the output of the forward pass activation function output (the tanh output) and running it through the linear layer with bias using the weights W_{hy}. Then, a softmax function is applied to the vector of outputs of this layer to turn them into probabilities of selection for the next character. A random character (actually a character index) is chosen with weights of selection for each character index equal to the probabilities from

the softmax output (so the higher probabilities correspond to higher chances of a certain character's index being selected). This index is selected and translated back into its corresponding character value to generate the next character. Then, the model continues running using the updated hidden state and most recently generated character to generate the next character through the forward pass / sampling process, and so on.

b) How does sampling work in the sample function?

Answer: The sampling function begins by selecting a character specified by the seed_idx parameter. This is treated as the first character in the string. The sampling function takes in the last value of the hidden state of the model (output by the model training loop) and uses this hidden state, combined with the specified character value, to run the forward pass of the RNN. The weights used during the forward pass are not specified as input into the function or within the function, so they appear to be the weights as of the last update in the training loop. A weighted draw is then taken from the softmax logit output of the forward pass, and the resulting integer selected identifies the index of the next character generated. This is then fed back into the RNN along with the updated hidden state to generate the next character, and so on, until n iterations are reached.

c) What happens if you modify the sampling temperature?

Answer: As noted in Sharma (2022), the temperature is a parameter in the softmax function, represented by T in the softmax expression:

$$s_i = \frac{e^{\frac{x_i}{T}}}{\sum_j e^{\frac{x_j}{T}}}$$

where s_i is the ith entry of the softmax output vector. Here, if $T = 1$, the softmax function behaves as usual. For very large T, the values in the numerator become closer to each other for different values of i, since the exponent is smaller (and thus $e^{\frac{x_i}{T}}$ changes relatively little in response to a change in x_i) (Sharma, 2022). Therefore, the softmax output vector (containing probabilities of selecting each character next) has entries that are more similar, resulting in a wider variety of characters selected, but less consistency, as noted by Karpathy (2015). For very low temperatures, the exponents are amplified and differences between them become more significant for the same differences in x_i (Sharma, 2022). Therefore, the model is highly likely to select the next letter that it has the most confidence is correct (even if this confidence is only slightly higher than other options). It tends to produce frequently-occurring output or patterns from the source text but does not produce more complex or creative strings, as Karpathy (2015) notes.

Modifications and Experiments

I was not able to get the model to train quickly enough on the Derecho resources (it is still taking on the order of multiple minutes per training iteration), and therefore have not been able to do sensitivity tests or adjusting parameters with the PyTorch RNN / LSTM models yet (Questions 1-4).

#1

#2

#3

#4

#5

#6

References

Prince, S. (2024). Understanding Deep Learning. The MIT Press. http://udlbook.com

Karpathy, A. (2015, May 21). The Unreasonable Effectiveness of Recurrent Neural Networks. Andrej Karpathy Blog. https://karpathy.github.io/2015/05/21/rnn-effectiveness/

Sharma, H. (2022, July 15). Softmax Temperature. Medium. https://medium.com/@harshit158/softmax-temperature-5492e4007f71