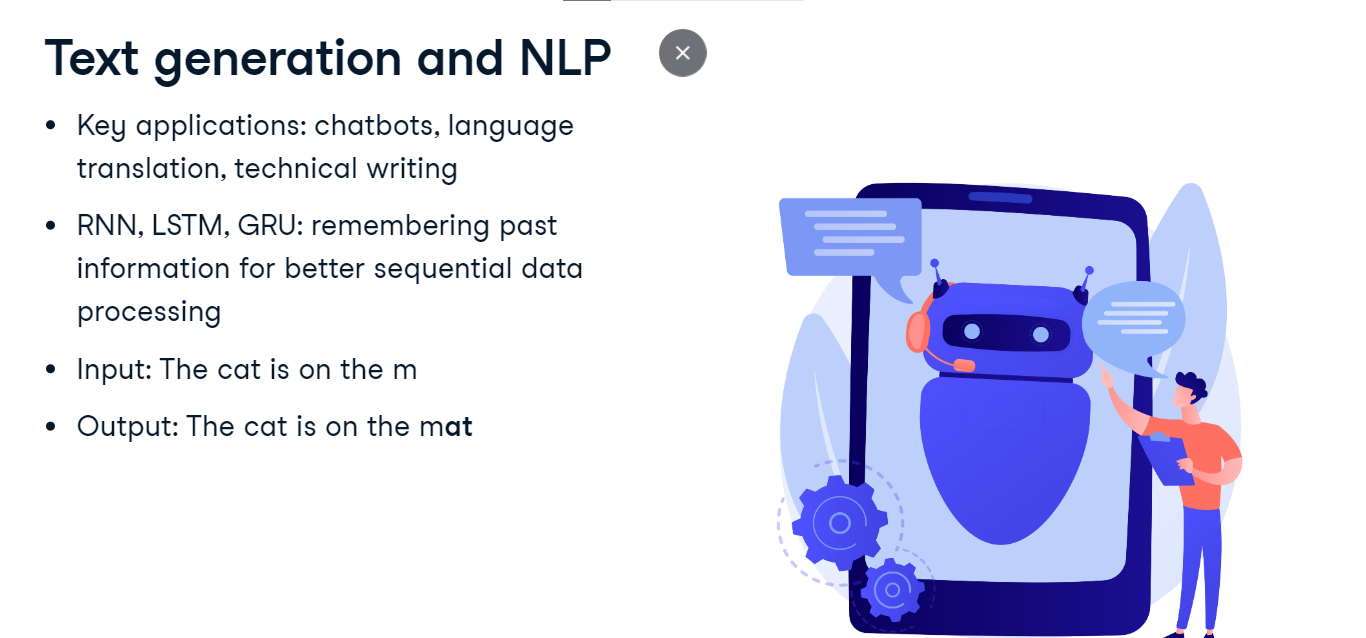
**Introduction to text generation**

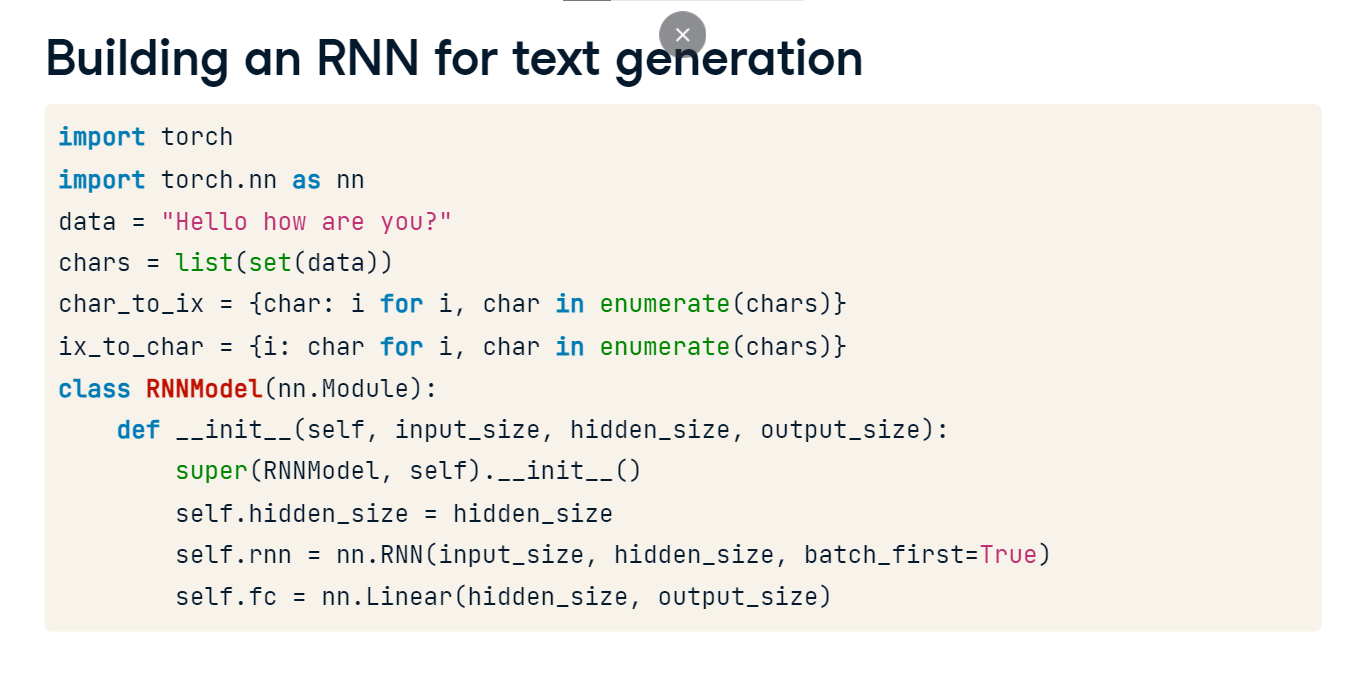
Let's discuss text generation with recurrent neural networks.

**Text generation and NLP**

Text generation is utilized in Natural Language Processing, serving various applications like chatbots, translations, and technical writing. RNNs, LSTMs, and GRUs ability to remember past information positions them as a key technology for processing sequential text data. For example, given the input "The cat is on the m", an RNN would complete the statement with "at", for "mat".

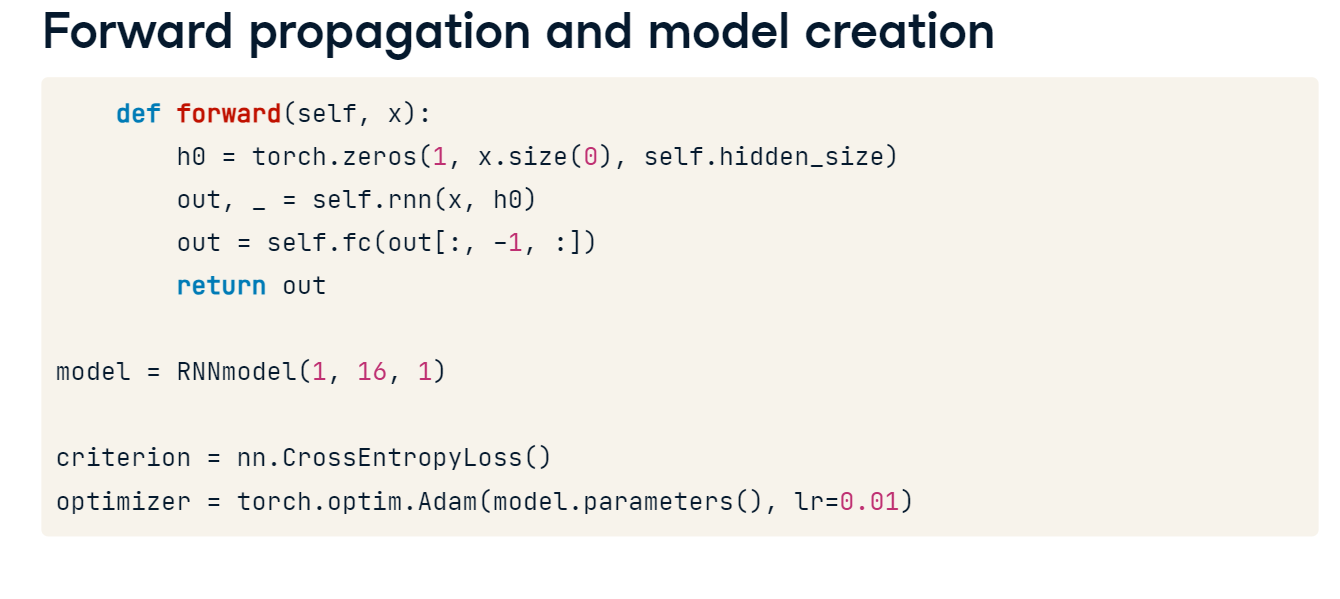


**Building an RNN for text generation**

To construct an RNN text generation model, we import the essential libraries: torch and nn. Initiating a "Hello how are you?" data variable, we extract unique characters and establish bidirectional mappings (character to index and vice versa) to convert textual data to numerical form, and subsequently back to text post-prediction. Character-to-integer conversions are favored over words for reduced dimensionality and computational ease, especially in smaller datasets, ensuring the model efficiently processes numerical input and generates readable text outputs. The init method takes input size, hidden size, and output size. Hidden size is stored as an instance variable. An RNN layer with specified dimensions is defined, followed by a fully connected layer. Our goal will be to train a model to generate "Hello how are you?". In training to predict subsequent characters, inputting 'h' should suggest 'e' as a likely next character if "he" is a frequent bigram.

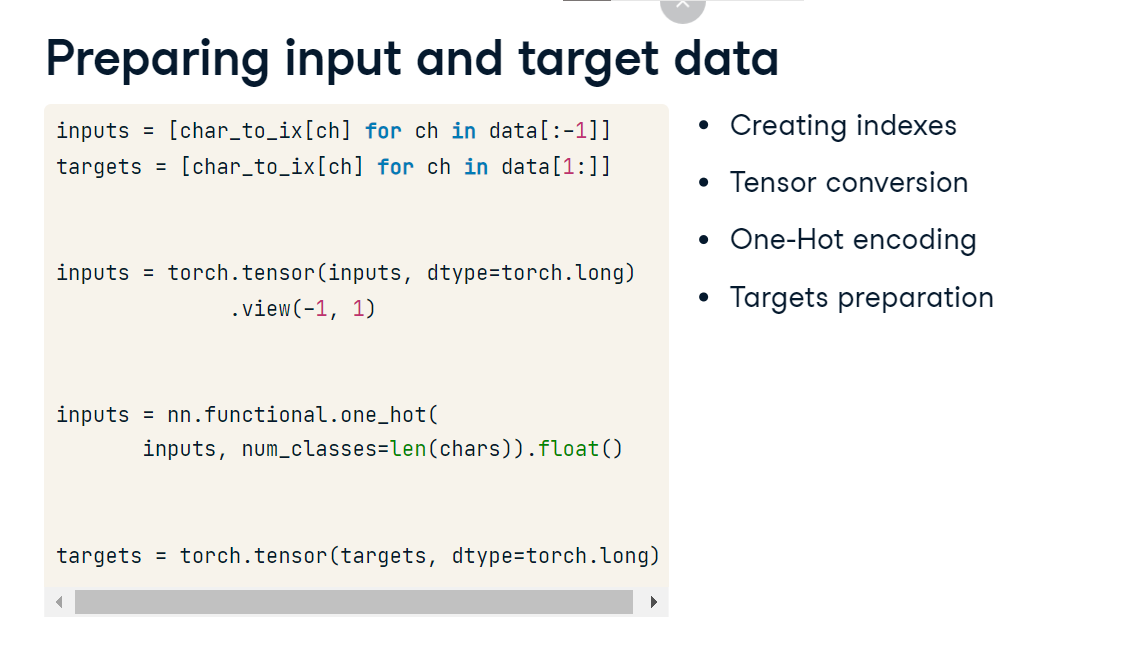
**Forward propagation and model creation**

In the forward method, we initialize a tensor of zeros for the initial hidden state, providing a neutral starting point for the RNN to learn from the data. We then pass the input data and the initial hidden state into the RNN layer to generate an output sequence and a new hidden state. We extract the last time step's output from the RNN and process it through a fully connected layer. This step allows us to convert the RNN's output into a format suitable as the next element in a generated sequence. We return this output as the result of the forward method. We instantiate our RNNmodel with an input size of one, a hidden size of 16, and an output size of one, optimized for single-token input and output sequences while allowing a 16-unit hidden layer for feature extraction. We treat text generation as a regression problem to predict the next token in a sequence as the next token can have infinite tensor output classes rather than a set number of output classes needed for classification. Our loss function is CrossEntropyLoss. We use the Adam optimizer, specifying the learning rate as 0.01.



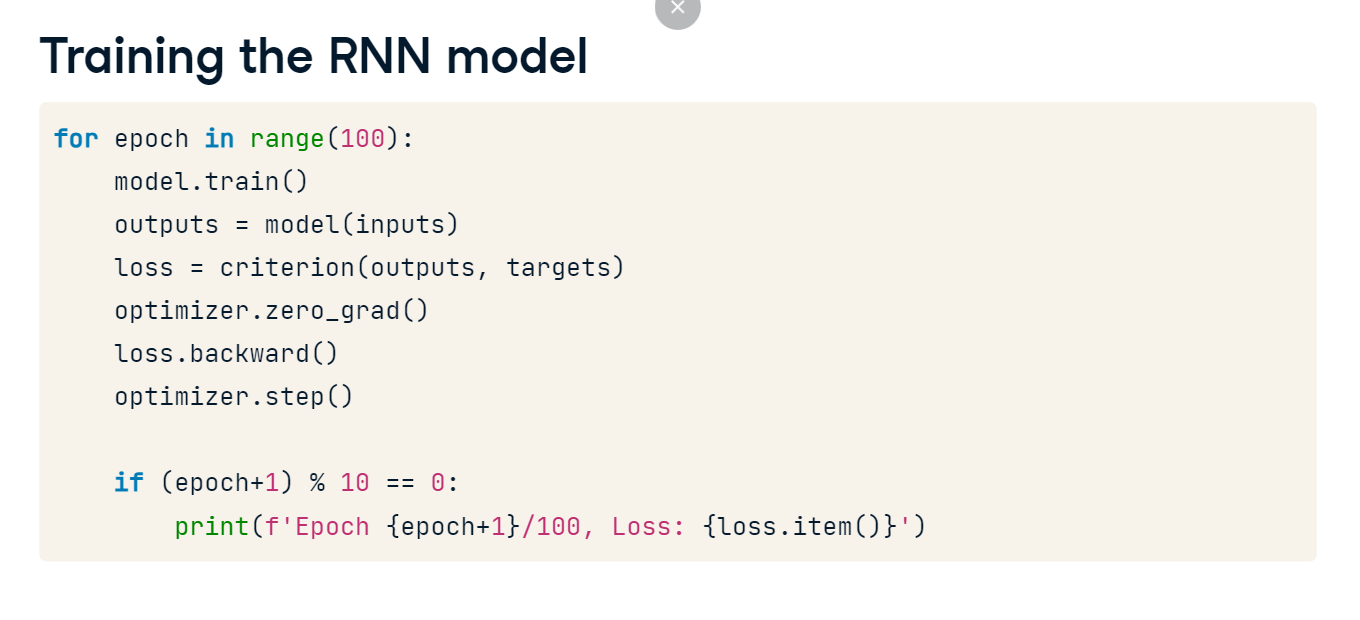
**Preparing input and target data**

The inputs and targets lists are created by mapping each character in the data string to its corresponding index, excluding the last character for inputs and the first character for targets. The index lists are then converted into long tensors. Additionally, we reshape inputs to have an additional dimension and match the expected input shape for the model. The inputs tensor is one-hot encoded, turning each index into a binary vector, where all elements are zero except for the one at the position of the index. However, the targets tensor remains as character indices to align with CrossEntropyLoss, which requires class indices as targets.



**Training the RNN model**

We initiate our training loop for 100 epochs, switch our model to training mode, and feed the inputs to the model and get the outputs. We calculate the loss by comparing the model's outputs to the actual targets. As PyTorch accumulates gradients, we clear the existing gradients in the optimizer. We perform backpropagation to compute loss gradients for the model parameters, and update them. Finally, we print the epoch number and the current loss every ten epochs. The output is shown on the next slide.



**Testing the model**

Let's test our trained RNN model. We switch the model to evaluation mode. We will prepare the character 'h' for prediction. 'h' is converted to its index using character to index mapping. The nn-dot-functional-dot-one\_hot function is used to one hot encode the index. The tensor is reshaped to a compatible format for encoding, with num\_classes set to the length of unique characters and converted to a float tensor. We feed test\_input into the model to get the predicted\_output. Using torch-dot-argmax on this output, we find the index of the maximum value along axis one, representing the most probable next character. We then print the model's prediction for this input. The decreasing loss over 100 epochs suggests our model learned and improved. When we input h into our trained model, it predicted e, which is fairly close. This indicates that our model has learned to generate text well.

