**Handling sequences with PyTorch**

We've learned to handle tabular and image data. Let's now discuss sequential data.

**Sequential data**

Sequential data is ordered in time or space, where the order of the data points is important and can contain temporal or spatial dependencies between them. Time series, data recorded over time like stock prices, weather, or daily sales is sequential. So is text, in which the order of words in a sentence determines its meaning. Another example is audio waves, where the order of data points is crucial to the sound reproduced when the audio file is played.

**Electricity consumption prediction**

In this chapter, we will tackle the problem of predicting electricity consumption based on past patterns. We will use a subset of the electricity consumption dataset from the UC Irvine Machine Learning Repository. It contains electricity consumption in kilowatts, or kW, for a certain user recorded every 15 minutes for four years.

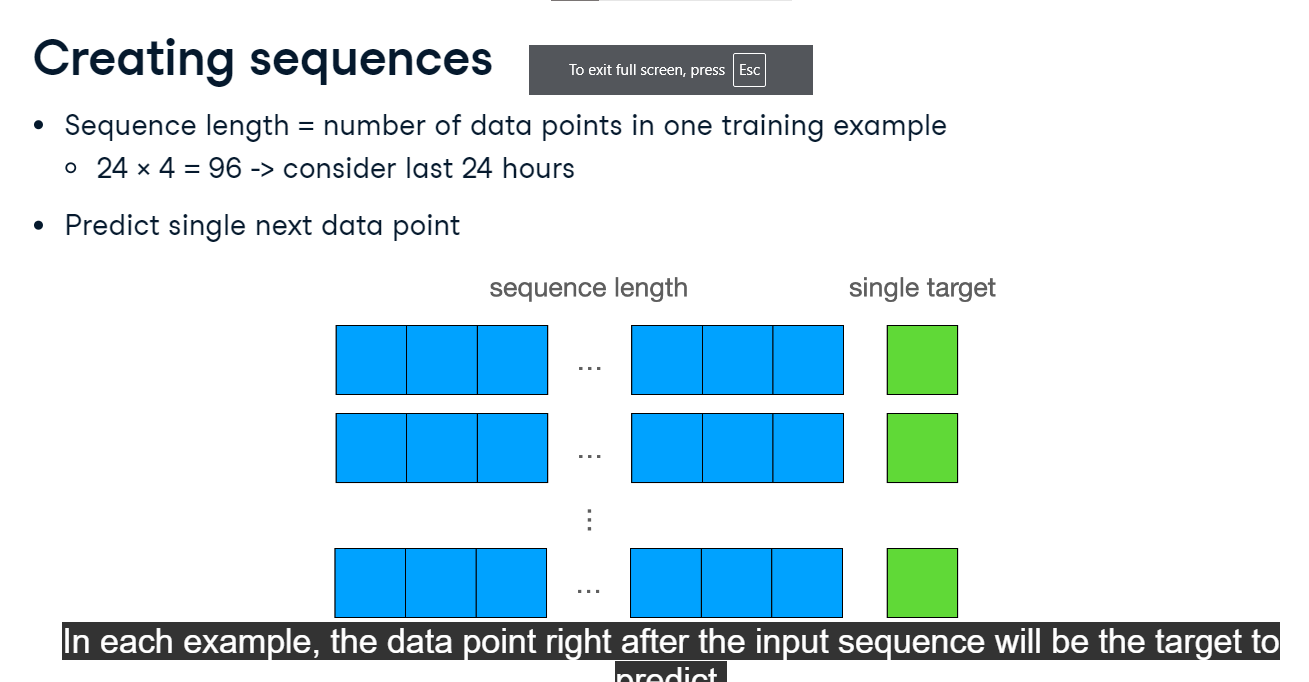
1. 1 Trindade,Artur. (2015). ElectricityLoadDiagrams20112014. UCI Machine Learning Repository. https://doi.org/10.24432/C58C86.

**Train-test split**

In many machine learning applications, one randomly splits the data into training and testing sets. However, with sequential data, there are better approaches. If we split the data randomly, we risk creating a look-ahead bias, where the model has information about the future when making forecasts. In practice, we won't have information about the future when making predictions, so our test set should reflect this reality. To avoid the look-ahead bias, we should split the data by time. We will train on the first three years of data, and test on the fourth year.

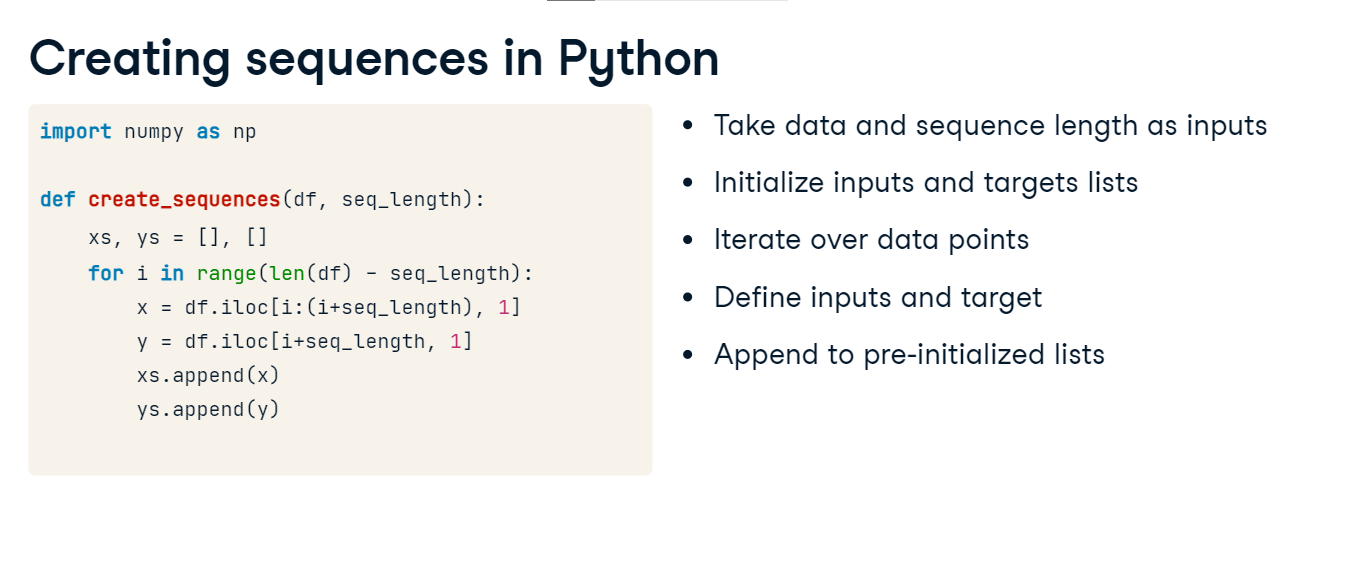
**Creating sequences**

To feed the training data to the model, we need to chunk it first to create sequences that the model can use as training examples. First, we need to select the sequence length, which is the number of data points in one training example. Let's make each forecast based on the previous 24 hours. Because data is at 15 minute intervals, we need to use 24 times 4 which is 96 data points. In each example, the data point right after the input sequence will be the target to predict.



**Creating sequences in Python**

Let's implement a Python function to create sequences. It takes the DataFrame and the sequence length as inputs. We start with initializing two empty lists, xs for inputs and ys for targets. Next, we iterate over the DataFrame. The loop only goes up to "len(df) - seq\_length", ensuring that for every iteration, there are always seq\_length data points available in the DataFrame for creating the sequence and a subsequent data point to serve as the target. For each considered data point, we define inputs x as the electricity consumption values starting from this point plus the next sequence length points, and the target y as the subsequent electricity consumption value. The 1 passed to the iloc method stands for the second DataFrame column, which stores the electricity consumption data. Finally, we append the inputs and the target to pre-initialized lists, and after the loop, return them as NumPy arrays.



**TensorDataset**

Let's use our function to create sequences from the training data. This gives us almost 35 thousand training examples. To convert them to a torch Dataset, we will use the TensorDataset function. We pass it two arguments, the inputs and the targets. Each argument is the NumPy array converted to a tensor with torch.from\_numpy and parsed to float. The TensorDataset behaves just like all other torch Datasets and it can be passed to a DataLoader in the same way.



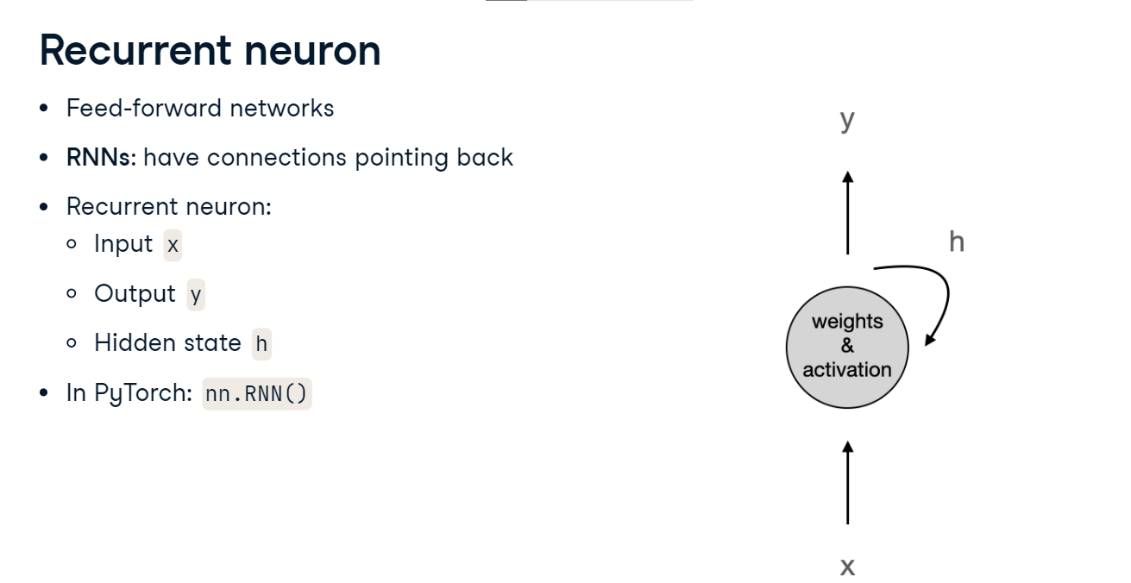
**Applicability to other sequential data**

Everything we have learned here can also be applied to other sequential data. For example, Large Language Models are trained to predict the next word in a sentence, a problem similar to predicting the next amount of electricity used. For speech recognition, which means transcribing an audio recording of someone speaking to text, one would typically use the same sequence-processing model architectures we will learn about soon.

**Recurrent Neural Networks**

**Recurrent neuron**

So far, we built feed-forward neural networks where data is passed in one direction: from inputs, through all the layers, to the outputs. Recurrent neural networks, or RNNs, are similar, but also have connections pointing back. At each time step, a recurrent neuron receives some input x, multiplied by the weights and passed through an activation. Out come two values: the main output y, and the hidden state, h, that is fed back to the same neuron. In PyTorch, a recurrent neuron is available as nn.RNN.

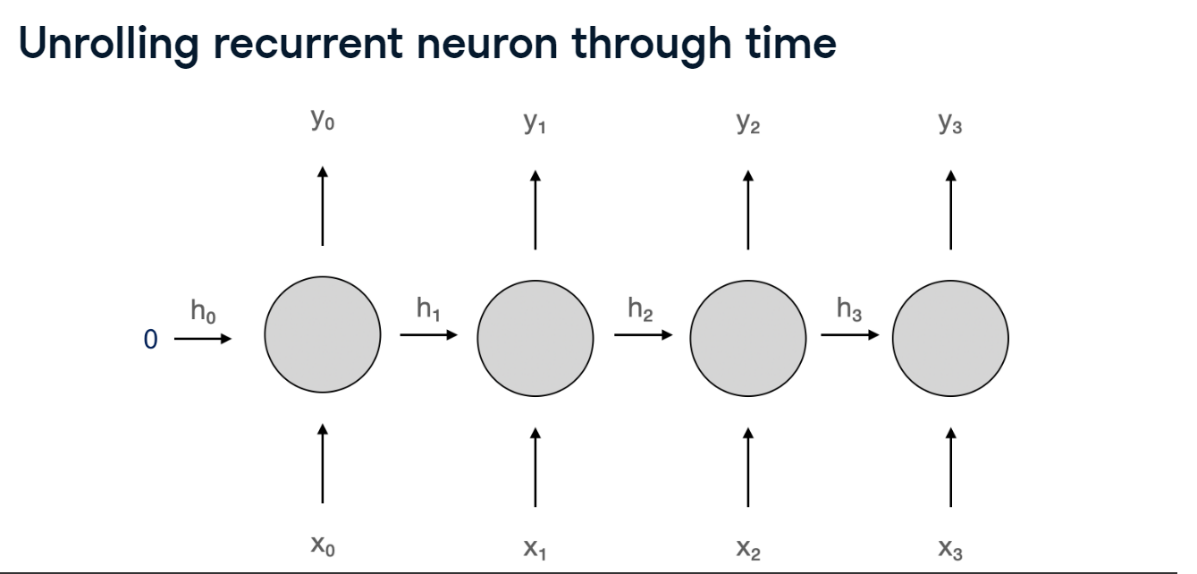


**Unrolling recurrent neuron through time**

We can represent the same neuron once per time step, a visualization known as unrolling a neuron through time. At a given time step, the neuron represented as a gray circle receives input data x-zero and the previous hidden state h0 and produces output y-zero and a hidden state h1.

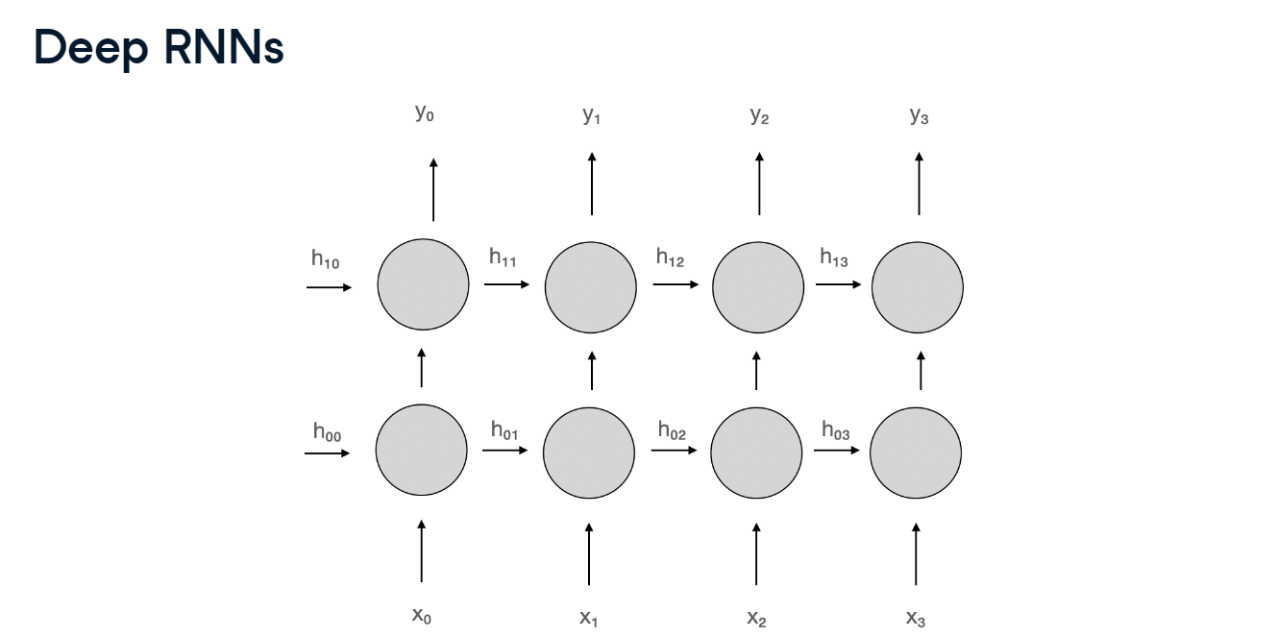
At the next time step, it takes the next value x1 as input and its last hidden state, h1.

And so it continues until the end of the input sequence. Since at the first time step there is no previous hidden state, h0 is typically set to zero. Notice that the output at each time step depends on all the previous inputs. This allows recurrent networks to maintain memory through time, which allows them to handle sequential data well.



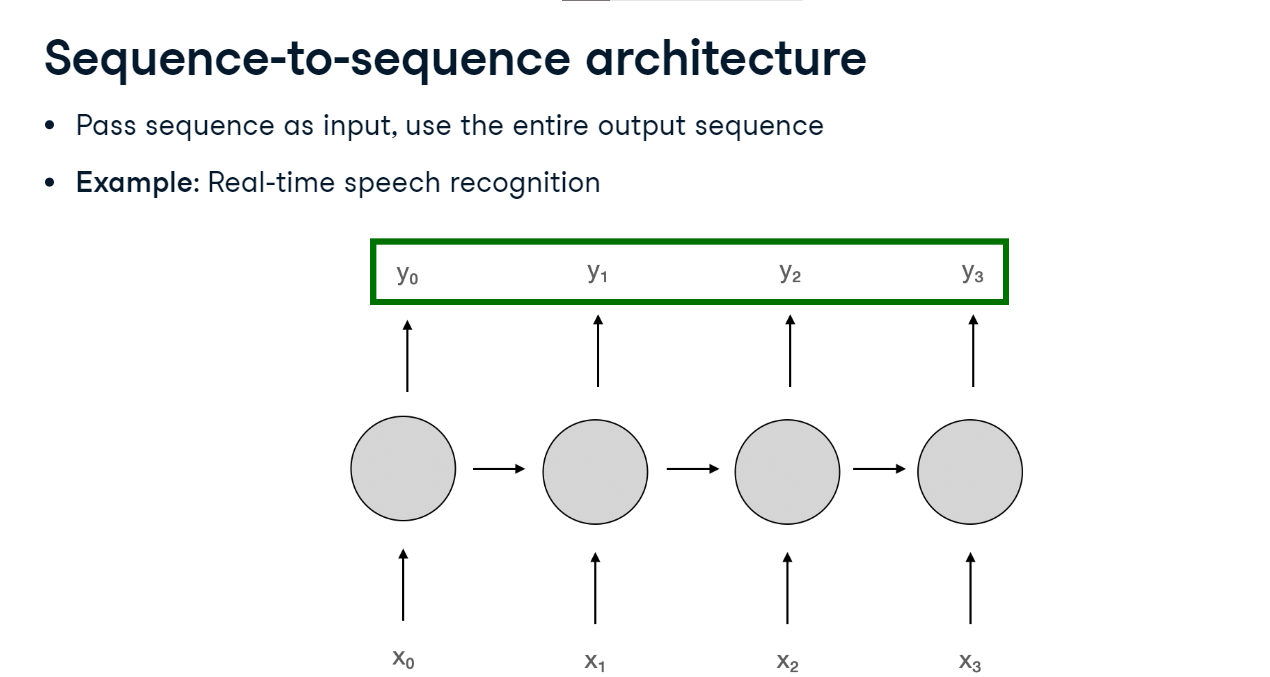
**Deep RNNs**

We can also stack multiple layers of recurrent cells on top of each other to get a deep recurrent neural network. In this case, each input will pass through multiple neurons one after another, just like in dense and convolutional networks we have discussed before

.

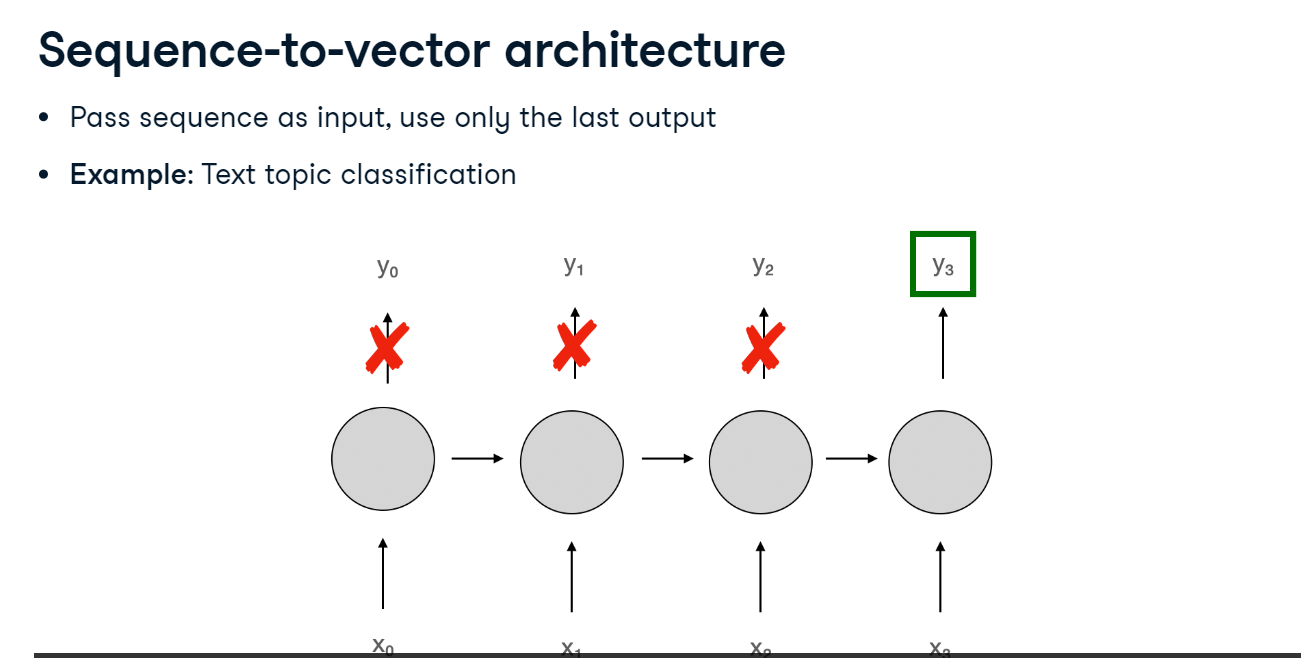
**Sequence-to-sequence architecture**

Depending on the lengths of input and output sequences, we distinguish four different architecture types. Let's look at them one by one. In a sequence-to-sequence architecture, we pass the sequence as input and make use of the output produced at every time step. For example, a real-time speech recognition model could receive audio at each time step and output the corresponding text.



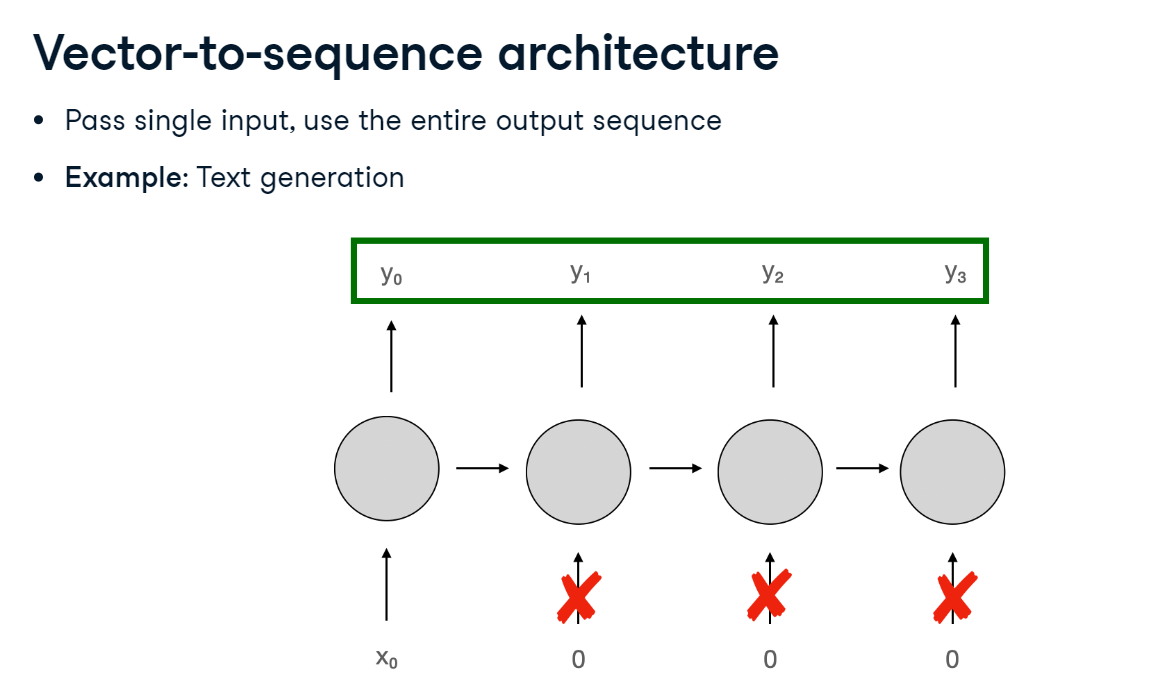
**Sequence-to-vector architecture**

In a sequence-to-vector architecture, we pass a sequence as input, but ignore all the outputs but the last one. In other words, we let the model process the entire input sequence before it produces the output. We can use this architecture to classify text as one of multiple topics. It's a good idea to let the model "read" the whole text before it decides what it's about. We will also use the sequence-to-vector architecture for electricity consumption prediction.



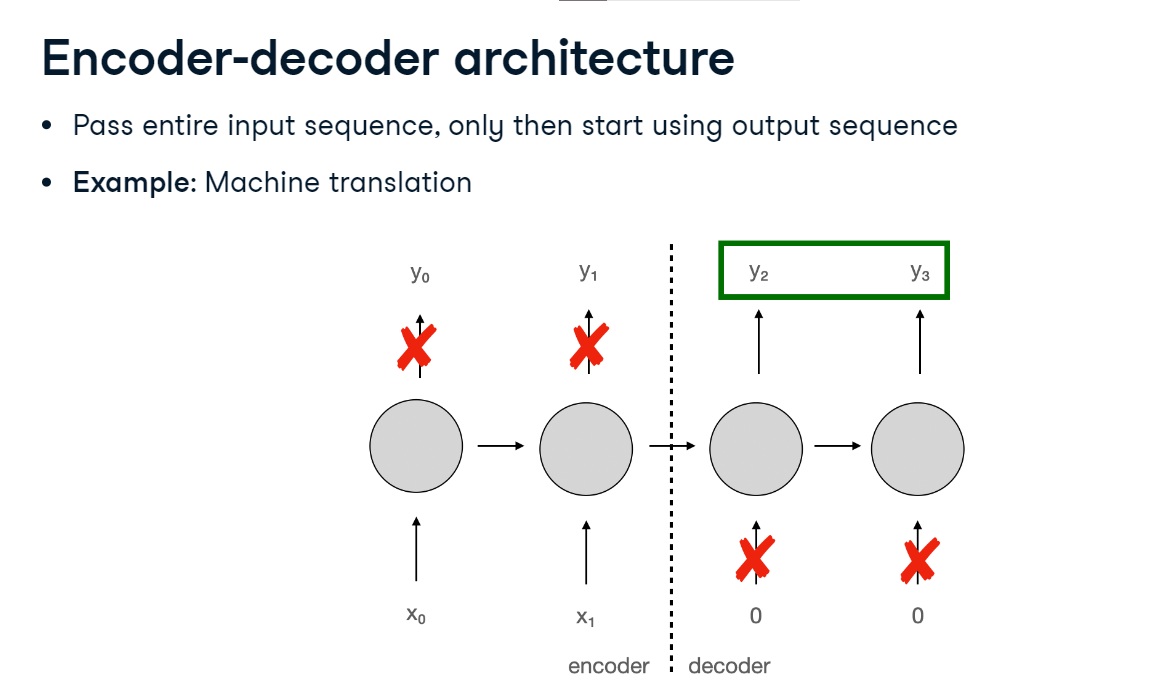
**Vector-to-sequence architecture**

One can also build a vector-to-sequence architecture where we pass a single input and replace all other inputs with zeros but make use of all the outputs from each time step. This architecture can be used for text generation: given a single vector representing a specific topic, style, or sentiment, a model can generate a sequence of words or sentences.



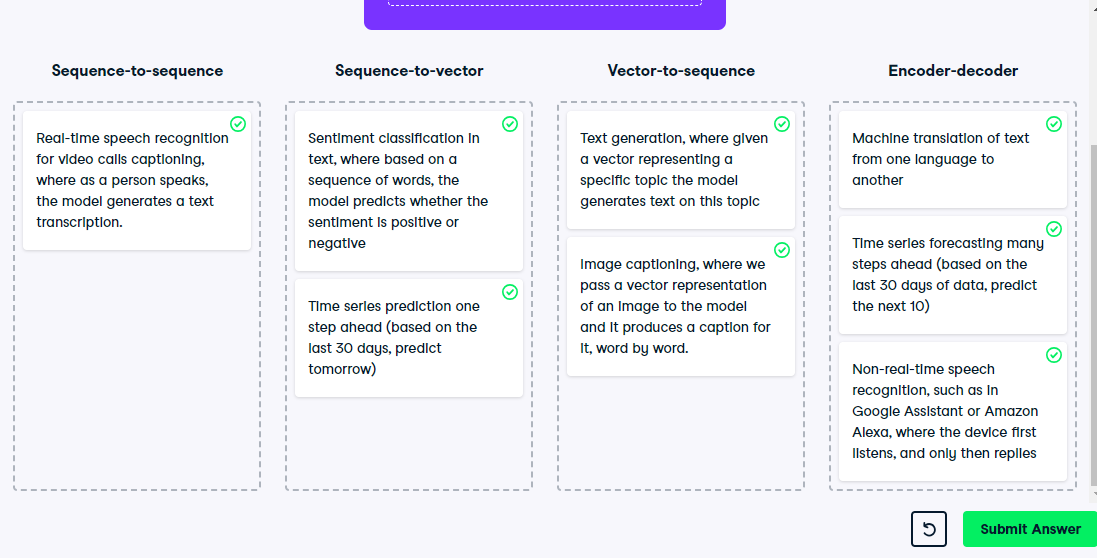
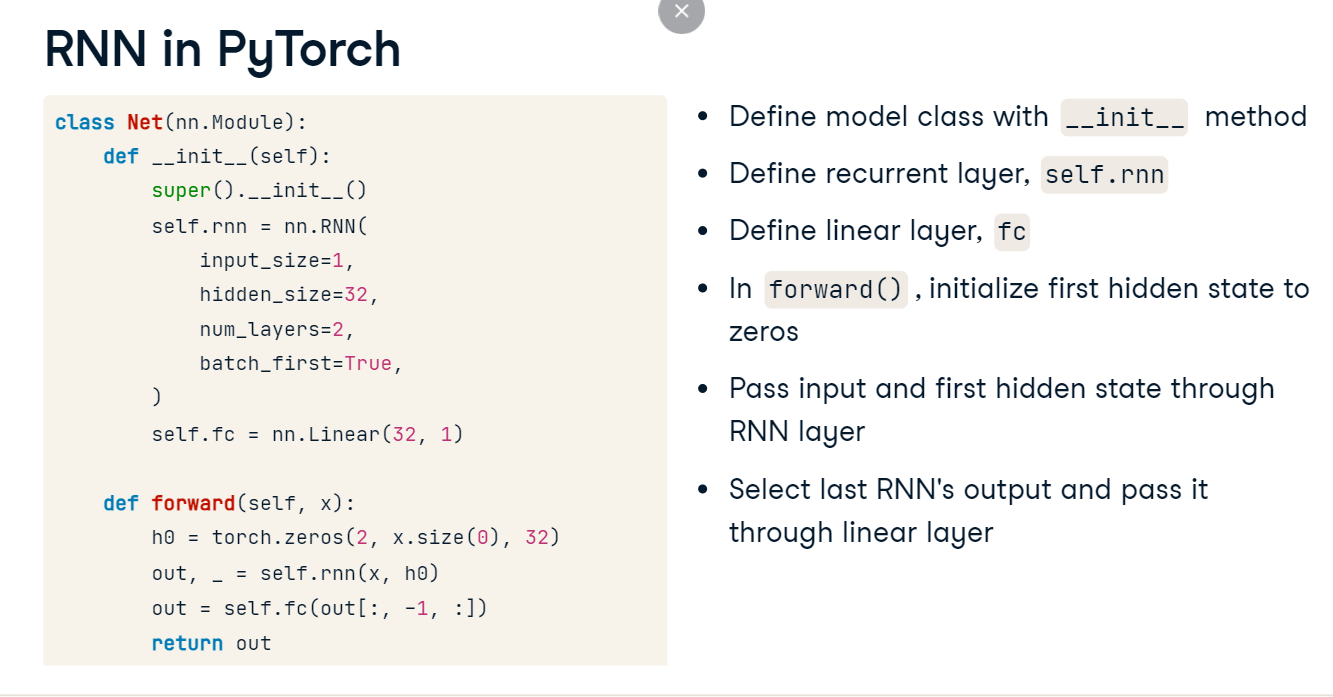
**Encoder-decoder architecture**

Finally, in an encoder-decoder architecture, we pass the input sequences, and only then start using the output sequence. This is different from sequence-to-sequence in which outputs are generated while the inputs are still being received. A canonical use case is machine translation. One cannot translate word by word; rather the entire input must be processed before output generation can start.



**RNN in PyTorch**

Let's build a sequence-to-vector RNN in PyTorch. We define a model class with the init method as usual. Inside it, we assign the nn.RNN layer to self.rnn, passing it an input size of 1 since we only have one feature, the electricity consumption, an arbitrarily chosen hidden size of 32 and 2 layers, and we set batch\_first to True since our data will have the batch size as its first dimension. We also define a linear layer mapping from the hidden size of 32 to the output of 1. In the forward method, we initialize the first hidden state to zeros using torch.zeros and assign it to h0. Its shape is the number of layers (2) by input size, which we extract from x as x.size-zero, by hidden state size (32). Next, we pass the input x and the first hidden state through the RNN layer. Then, we select only the last output by indexing the middle dimension with -1, pass the result through the linear layer, and return.



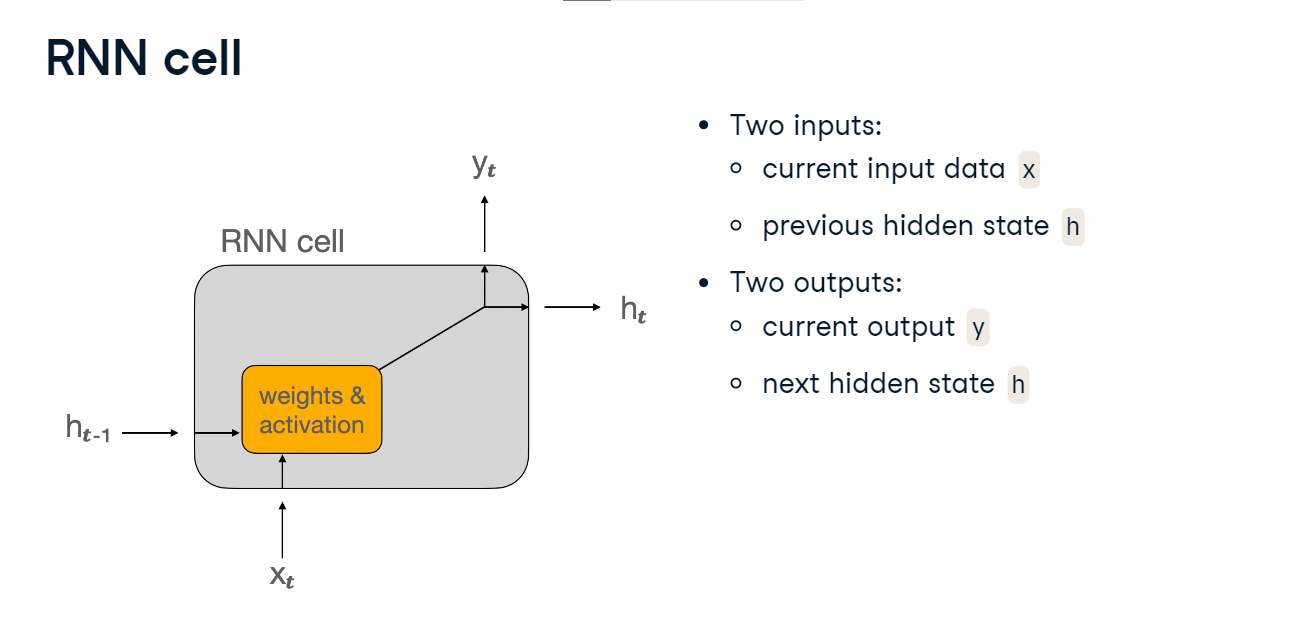
**LSTM and GRU cells**

**Short-term memory problem**

Because RNN neurons pass the hidden state from one time step to the next, they can be said to maintain some sort of memory. That's why they are often called RNN memory cells, or just cells for short. However, this memory is very short-term: by the time a long sentence is processed, the hidden state doesn't have much information about its beginning. Imagine trying to translate a sentence between languages; as soon as we have read it, we don't remember how it started. To solve this short-term memory problem, two more powerful types of cells have been proposed: the Long Short-Term Memory or LSTM cell and the Gated Recurrent Unit or GRU cell.

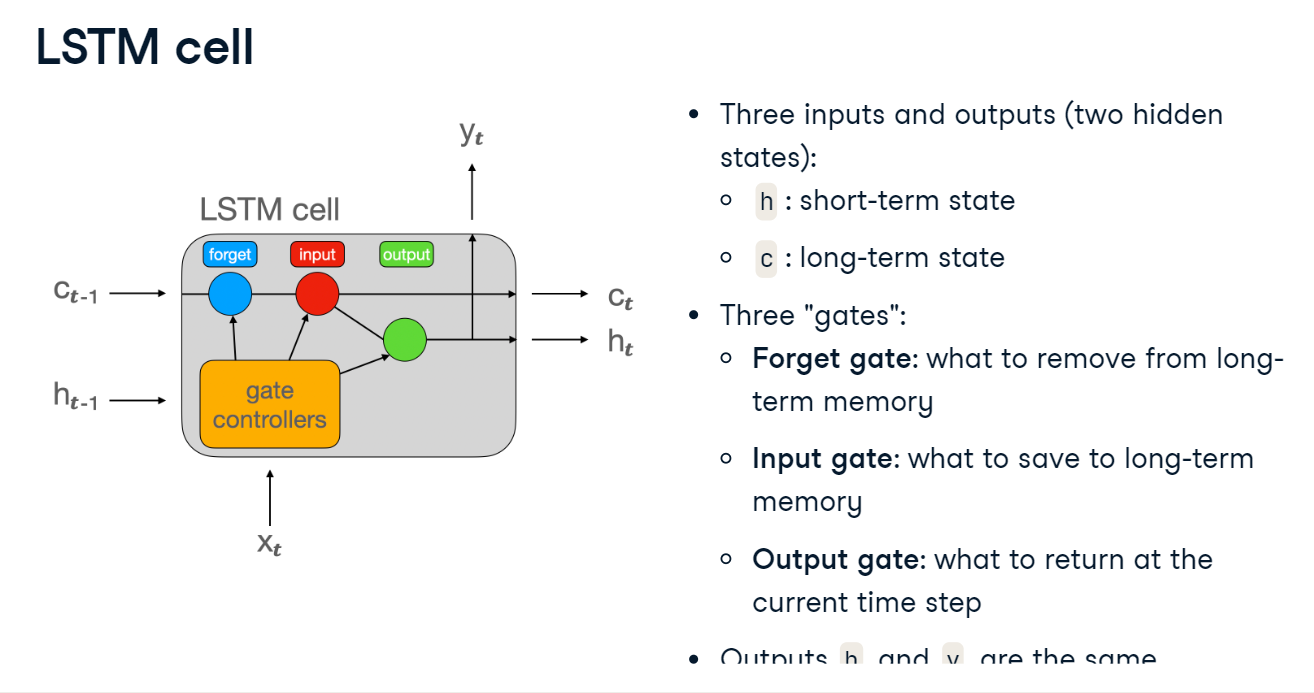
**RNN cell**

Before we look at LSTM and GRU cells, let's visualize the plain RNN cell. At each time step t, it takes two inputs, the current input data x and the previous hidden state h. It multiplies these inputs with the weights, applies activation, and outputs two things: the current outputs y and the next hidden state.



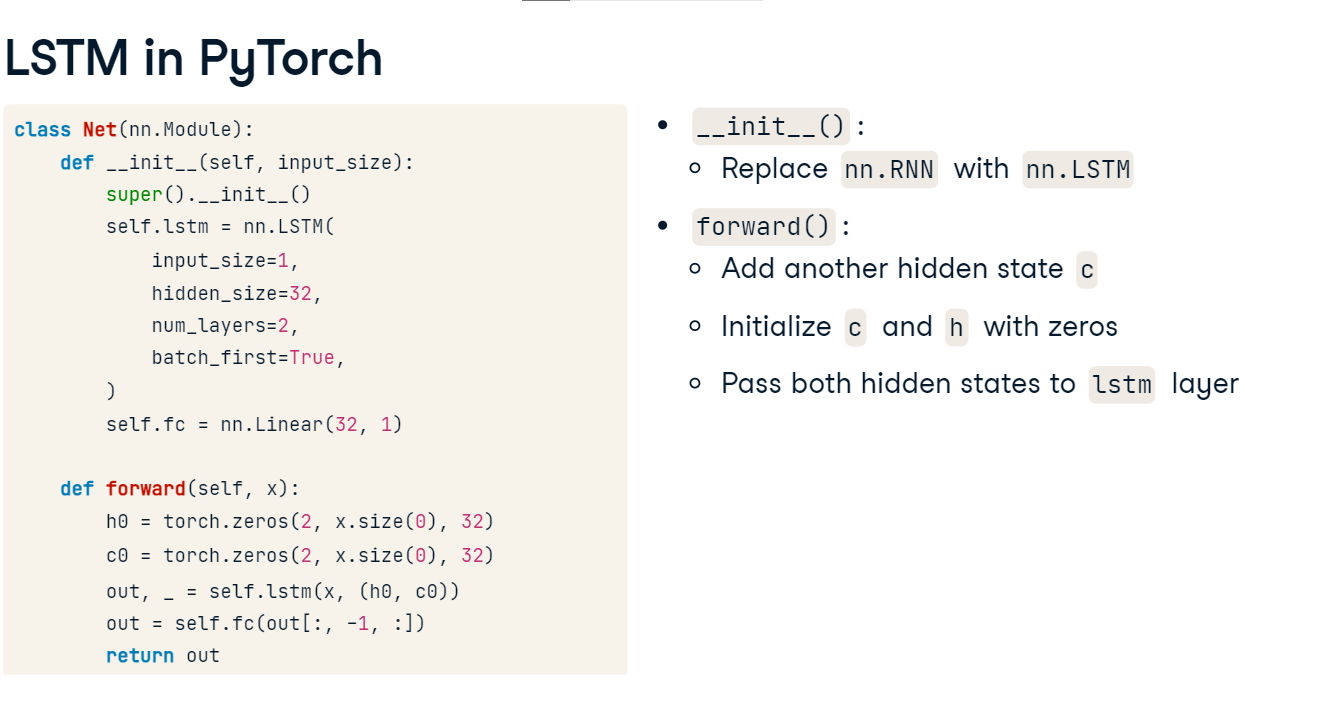
**LSTM cell**

The LSTM cell has three inputs and outputs. Next to the input data x, there are two hidden states: **h represents the short-term memory and c the long-term memory**. At each time step, h and x are passed through some linear layers called gate controllers which determine what is important enough to keep in the long-term memory. The gate controllers first erase some parts of the long-term memory in the forget gate. Then, they analyze x and h and store their most important parts in the long-term memory in the input gate. This long-term memory, c, is one of the outputs of the cell. At the same time, another gate called the output gate determines what the current output y should be. The short-term memory output h is the same as y.



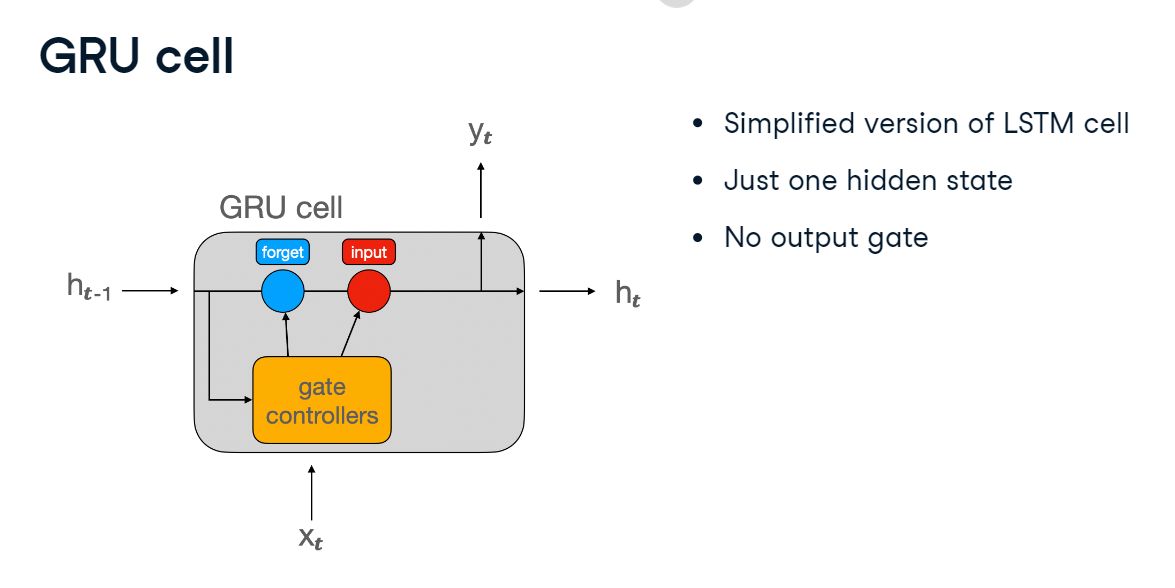
**LSTM in PyTorch**

Building an LSTM network in PyTorch is very similar to the plain RNN we have already seen. In the init method, we only need to use the nn.LSTM layer instead of nn.RNN. The arguments that the layer takes as inputs are the same. In the forward method, we add the long-term hidden state c and initialize both h and c with zeros. Then, we pass h and c as a tuple to the LSTM layer. Finally, we take the last output, pass it through the linear layer and return just like before.



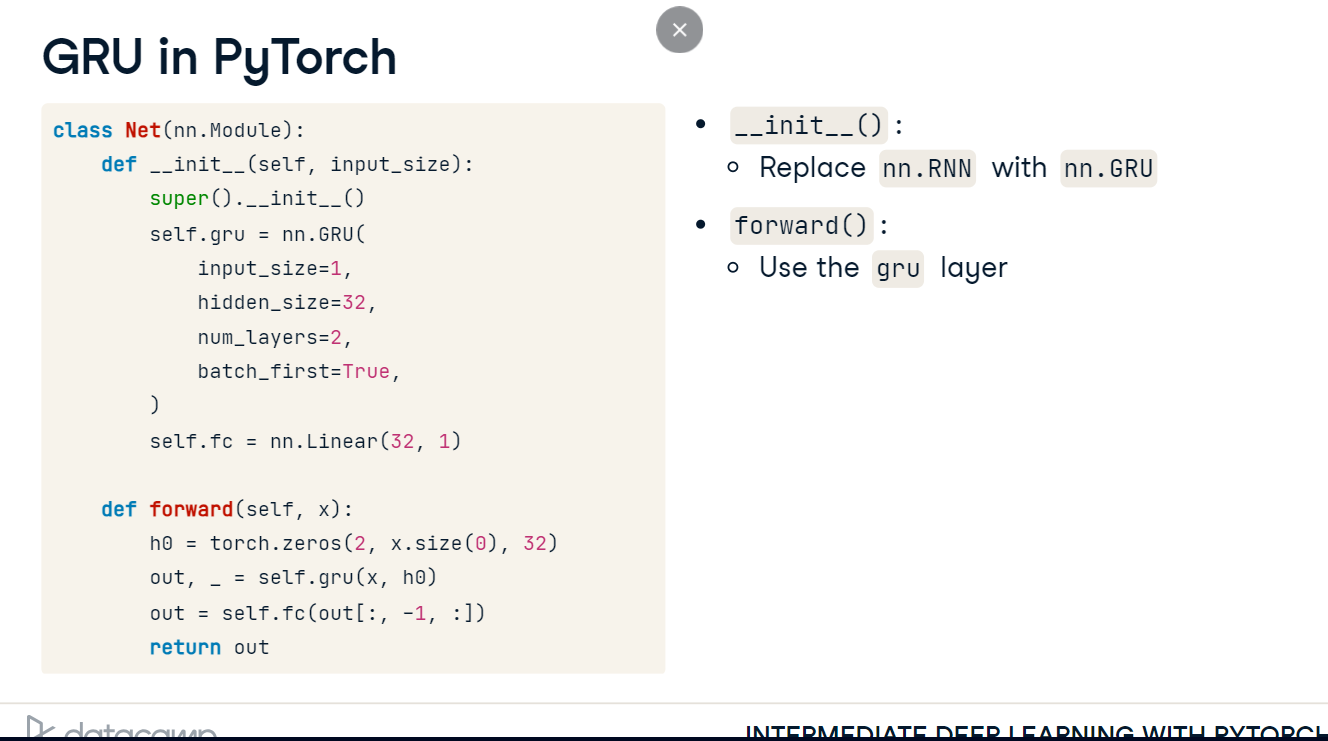
**6. GRU cell**

The GRU cell is a simplified version of the LSTM cell. It merges the long-term and short-term memories into a single hidden state. It also doesn't use an output gate: the entire hidden state is returned at each time step.



**GRU in PyTorch**

Building a GRU network in PyTorch is almost identical to the plain RNN. All we need to do is replace the nn.rnn with nn.gru when defining the layer in the init method, and then call the new gru layer in the forward method.



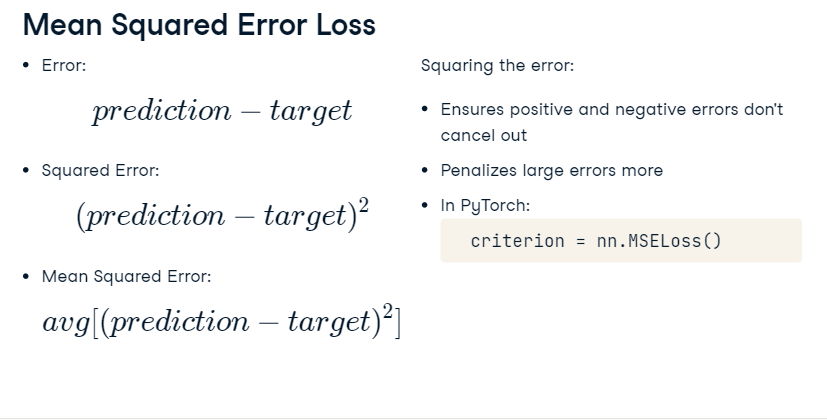
**Should I use RNN, LSTM, or GRU?**

So, which type of recurrent network should we use: the plain RNN, LSTM, or GRU? There is no single answer, but consider the following. Although plain RNNs have revolutionized modeling of sequential data and are important to understand, they are not used much these days because of the short-term memory problem. Our choice will likely be between LSTM and GRU. GRU's advantage is that it's less complex than LSTM, which means less computation. Other than that, the relative performance of GRU and LSTM varies per use case, so it's often a good idea to try both and compare the results. We will learn how to evaluate these models soon.

**Training and evaluating RNNs**

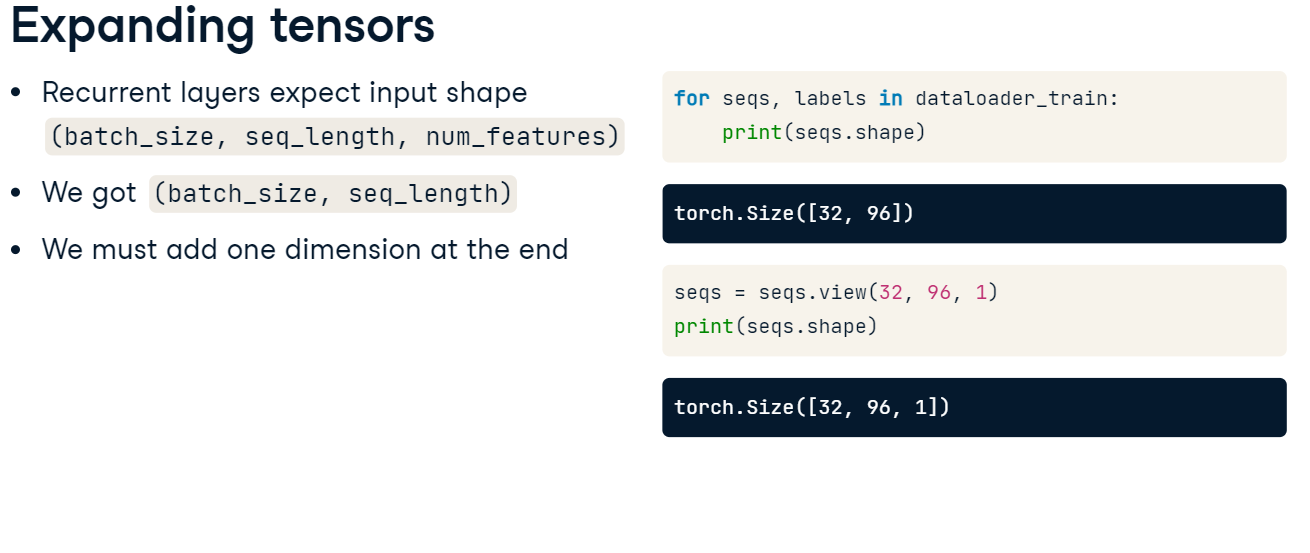
**Mean Squared Error Loss**

Up to now, we have been solving classification tasks using cross-entropy losses. Forecasting of electricity consumption is a regression task, for which we will use a different loss function: Mean Squared Error. Here is how it's calculated. The difference between the predicted value and the target is the error. We then square it, and finally average over the batch of examples. Squaring the errors plays two roles. First, it ensures positive and negative errors don't cancel out, and second, it penalizes large errors more than small ones. Mean Squared Error loss is available in PyTorch as nn.MSELoss.



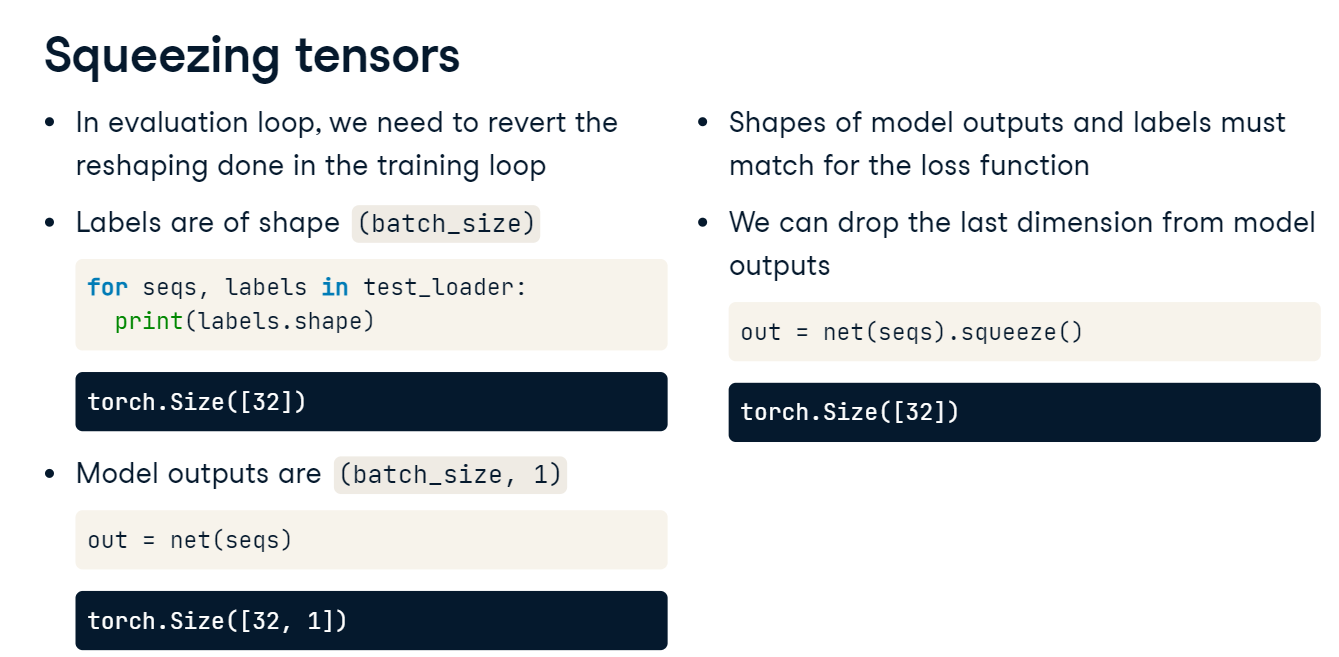
**Expanding tensors**

Before we take a look at the model training and evaluation, we need to discuss two useful concepts: expanding and squeezing tensors. Let's tackle expanding first. All recurrent layers, RNNs, LSTMs, and GRUs, expect input in the shape: batch size, sequence length, number of features. But as we loop over the DataLoader, we can see that we got the shape batch size of 32 by the sequence length of 96. Since we are dealing with only one feature, the electricity consumption, the last dimension is dropped. We can add it, or expand the tensor, by calling view on the sequence and passing the desired shape.



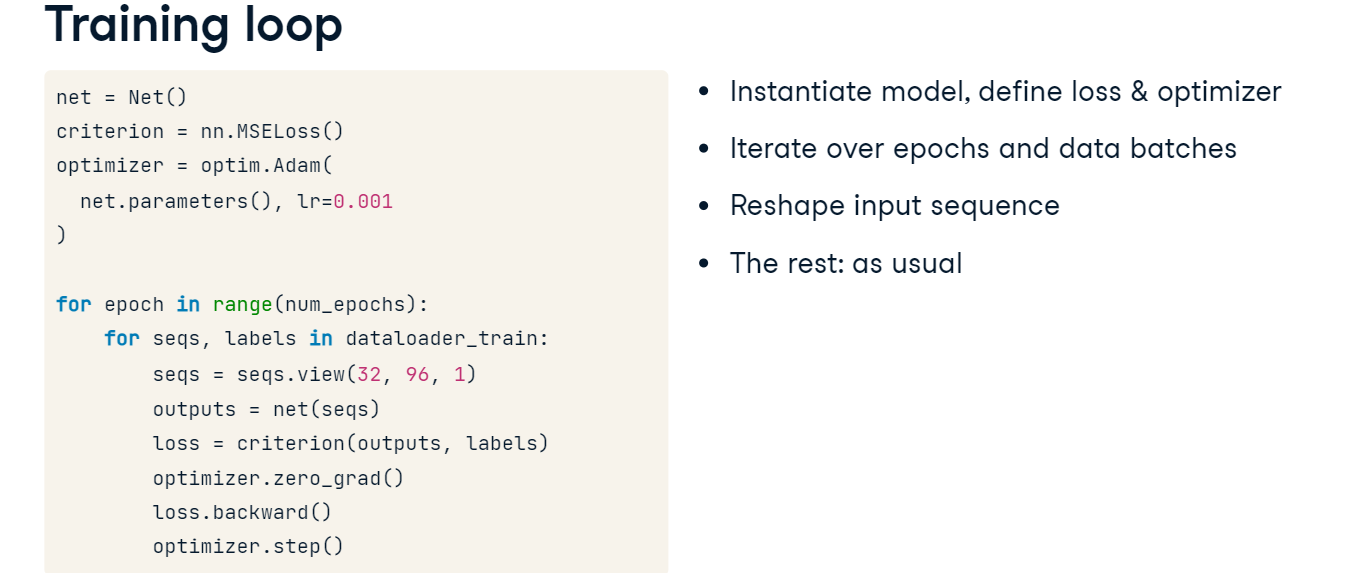
**Squeezing tensors**

Conversely, as we evaluate the model, we will need to revert the expansion we have applied to the model inputs which can be achieved through squeezing. Let's see why that's the case and how to do it. As we iterate through test data batches, we get labels in shape batch size. Model outputs, however, are of shape batch size by 1, our number of features. We will be passing the labels and the model outputs to the loss function, and each PyTorch loss requires its inputs to be of the same shape. To achieve that, we can apply the squeeze method to the model outputs. This will reshape them to match the labels' shape.



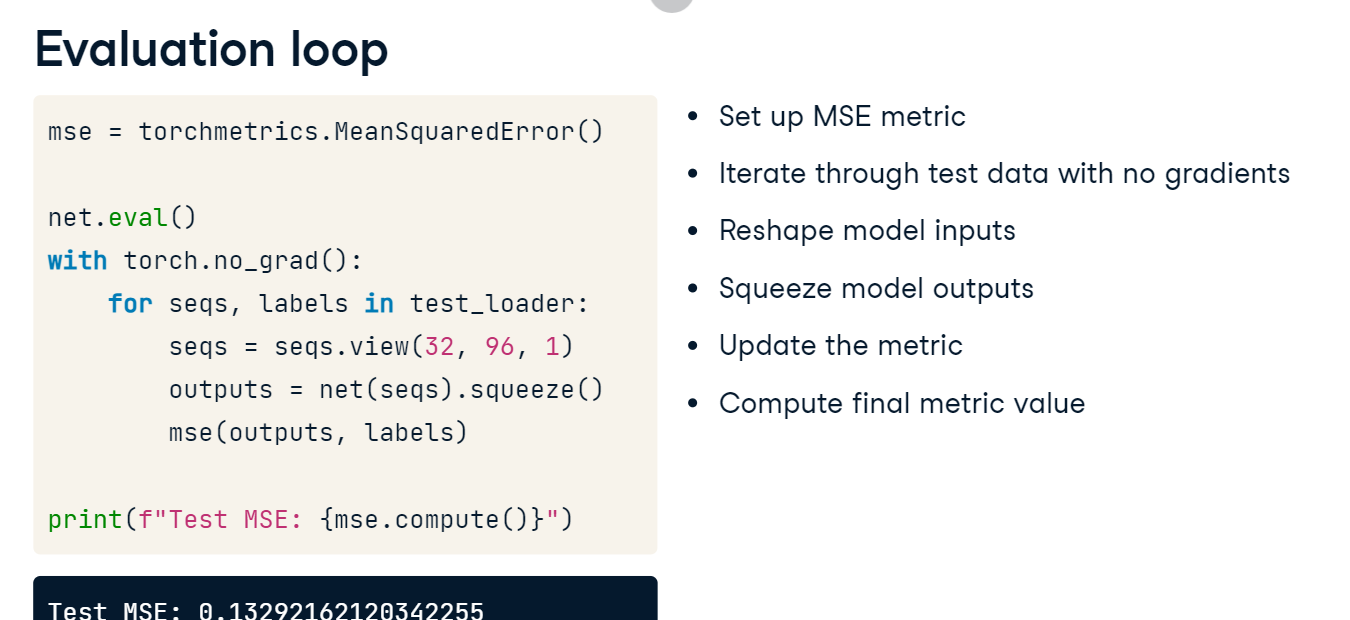
**Training loop**

The training loop is similar to what we have already seen. We instantiate the model and define the loss and the optimizer. Then, we iterate over epochs and training data batches. For each batch, we reshape the input sequence as we have just discussed. The rest of the training loop is the same as before.



**Evaluation loop**

Let's look at the evaluation loop. We start by setting up the Mean Squared Error metric from torchmetrics. Then, we iterate through test data batches without computing the gradients. Next, we reshape the model inputs just like during training, pass them to the model, and squeeze the outputs. Finally, we update the metric. After the loop, we can print the final metric value by calling compute on it, just like we did before.



**LSTM vs. GRU**

Here is our LSTM's test Mean Squared Error again. Let's see how it compares to a GRU network. It seems that for our electricity consumption dataset, with the task defined as predicting the next value based on the previous 24 hours of data, both models perform similarly, with GRU achieving even a slightly lower error. In this case, GRU might be preferred as it achieves the same or better results while requiring less processing power.

