10.6. Recurrent neural networks

Suppose we're given a dataset of strings, and we want to be able to generate new strings of the same general type.

aaron	abraham	admatha
abdiel	achaia	adria
abel	adam	ahab
abigail	adlai	...

Let's write a string x as a sequence of characters, $x = x_1 x_2 \cdots x_n$. We want to be able to generate random sequences $X = X_1 X_2 \cdots X_N$ where both the characters and the length are random. We can take exactly the same approach we've taken to every single probability modelling problem throughout this course: first invent a parameterized distribution $\Pr_X(x\,;\,\theta)$, then fit the model, then use the fitted model to generate new random sequences. The quality of the results will of course depend on the model we choose. No model is true—but some models fit the data better than others, and we can evaluate the fit in the usual way, by the log likelihood of holdout data.

OTHER PROBABILITY MODELS FOR SEQUENCES

The recurrent neural network can be thought of as a probability model for generating random sequences, designed to address shortcomings of other sequence models. It's useful to first review those other models. The simplest sequence model is a Markov chain.

Let's augment the alphabet with $\varnothing$ to denote 'start of string' and $\square$ to denote 'end of string', and generate random strings by starting at $\varnothing$ and jumping from character to character until we hit $\square$. The causal diagram is

$$\varnothing \to X_1 \to X_2 \to \cdots \to X_N \to \square$$

and the likelihood function is

$$\Pr_X(x_1 x_2 \cdots x_n\,;\,\theta) = p_\theta(x_1 \mid \varnothing)\, p_\theta(x_2 \mid x_1) \times \cdots \times p_\theta(\square \mid x_n)$$

where θ governs the character-to-character transition probabilities.

We'll get better results if we include some history, as per Markov's trigram model.

The causal diagram for Markov's trigram model, example 10.2.2, is

$$\begin{tikzcd}
\varnothing\varnothing \arrow[r, dotted] \arrow[d] & \varnothing X_1 \arrow[r, dotted] \arrow[d] & {} & \cdots & X_{N-2}X_{N-1} \arrow[r, dotted] \arrow[d] & X_{N-1}X_N \arrow[d] \\
X_1 \arrow[ur, dotted] & X_2 \arrow[ur, dotted] & & & X_N \arrow[ur, dotted] & \square
\end{tikzcd}$$

and the likelihood function is

$$\Pr_X(x_1 x_2 \cdots x_n\,;\,\theta) = p_\theta(x_1 \mid \varnothing\varnothing) p_\theta(x_2 \mid \varnothing x_1) \times \cdots \times p_\theta(\square \mid x_{n-1} x_n).$$

In this diagram the solid arrows denote random generation of new values, and the dotted arrows denote inputs to the deterministic function that updates the 'last two characters' state variable.

How much history should we keep? The more we keep, the more faithful our random sequences are to the training dataset—but if we're too faithful then we'll just regurgitate the training data rather than generate novel sequences, so we'll assign a low likelihood to the novel datapoints in the holdout set, so we'll score badly on evaluation. Rather than hard-coding a fixed history window, a flexible alternative is to use a hidden Markov model, where the hidden state is some abstract summary of everything that's worth remembering about the past.

$$\begin{tikzcd}
0 \arrow[r] & Z_1 \arrow[r] \arrow[d] & Z_2 \arrow[r] \arrow[d] & \cdots & Z_N \arrow[r] \arrow[d] & Z_{N+1} \arrow[d] \\
& X_1 & X_2 & & X_N & \square
\end{tikzcd}$$

In this particular version we've included an initial hidden state 0, to allow the first productive hidden state Z_1 to be random. The likelihood function is

$$\Pr_X(x_1 x_2 \cdots x_n\,;\,\theta) = \sum_{z_1,\dots,z_{n+1}} \left\{ \begin{array}{c} p_\theta(z_1|0)\, q_\theta(x_1|z_1) \times p_\theta(z_2|z_1)\, q_\theta(x_2|z_2) \times \cdots \\ \cdots \times p_\theta(z_{n+1}|z_n)\, q_\theta(\square|z_{n+1}). \end{array} \right\}$$

where p is the transition probabilities for the hidden state, and q is the emission probabilities for reading off characters, and θ represents all the parameters that need to be estimated for both these functions.

On one hand the hidden Markov is wonderfully flexible because we can choose the hidden state space however we like. On the other hand this is a useless idea because it doesn't solve anything, it just passes the buck on to whoever has to design the hidden state transitions; and because we've ended up with a horrible intractable sum for the likelihood function. This probability model will not be easy to work with when it comes to fitting the distribution.

RECURRENT NEURAL NETWORKS TO DIGEST HISTORY

It's useful to be able to keep track of the process's history, but it's hard to know exactly how history should be represented. What if we could train a neural network to learn this? That's exactly the idea behind the recurrent neural network probability model. It has the same causal diagram structure as the trigram model, but instead of a hard-coded history window it allows a flexible representation of history via a neural network and a 'history digest' s_i that can have as many dimensions as we like.

$$\begin{tikzcd}
0 \arrow[r, dotted] & \boxed{f_\theta}\; s_1 \arrow[r, dotted] \arrow[d] & \boxed{f_\theta}\; s_2 \arrow[r, dotted] \arrow[d] & \boxed{f_\theta}\; s_3 \arrow[d] & \cdots & s_N \arrow[r, dotted] \arrow[d] & \boxed{f_\theta}\; s_{N+1} \arrow[d] \\
\varnothing \arrow[ur, dotted] & X_1 \arrow[ur, dotted] & X_2 \arrow[ur, dotted] & X_3 & & X_N \arrow[ur, dotted] & \square
\end{tikzcd}$$

Here, f_θ is some neural network that takes in the latest event X_{i-1} and the previous history digest s_{i-1}, and computes a new history digest s_i. The next event X_i is then generated randomly, using s_i as its parameters. To save the bother of having to specify how X_i is to be generated, we might as well split s_i into two parts, $s_i = (p_i, v_i)$ where p_i specifies the likelihood of X_i and v_i is used for computing s_{i+1}. This leads to the basic recurrence equations

$$(p_i, v_i) = f_\theta(x_{i-1}, v_{i-1}), \qquad X_i \sim \text{Cat}(p_i).$$

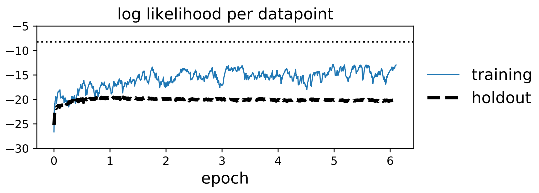
Generating random sequences. Once we've trained the neural network f_θ it's easy to generate a random string.

```
def generate():
    x,v = "$\varnothing$",0
    while x[-1] $\neq$ $\square$:
        p,v = $f_\theta$(x[-1], v)
        newchar = np.random.choice(ALPHABET, p)
        x = x + newchar
    return x[1:-1]   # strip out $\varnothing$ and $\square$
```

Training the network. The neural network is easily trained because, like the trigram model, there is an explicit formula for the log likelihood of a datapoint. The key is that the states are visible, not hidden—they're patent, not latent—in the sense that given a datapoint $x = x_1 x_2 \cdots x_n$ we can compute $s_1, \dots, s_n$. In pseudocode the log likelihood function looks like this:

```
def loglik(x):
    res = 0
    lastchar,v = $\varnothing$,0
    for char in x + "$\square$":
        p,v = $f_\theta$(lastchar, v)
        res = res + log(p[char])
        lastchar = char
    return res
```

Evaluating the model. The perfectly-fitted probability model for the training dataset is its empirical distribution, which assigns probability $1/N$ to every one of the N datapoints in the whole dataset (assuming no duplication). The best possible log likelihood we can possibly achieve during training is thus $N \log(1/N)$. This can give us a hint about whether our network is overfitting, or whether it's too simple to overfit:



- If the training loss gets close to this bound, it suggests the network is overfitted to the training data, and so the log likelihood of the holdout dataset will be low. This suggests we need to add a regularizing mechanism such as dropout.
- If the neural network can't get close to this even with the regularizing mechanism turned off, it suggests the network isn't complex enough to describe the dataset, so we need more nodes or layers.

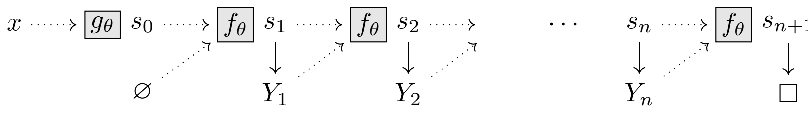
* ✻ *

Here is PyTorch code, a bit more fussy than our pseudocode. The convention for PyTorch sequence operations is to act on tensors of shape $n \times b \times f$ where we're working on a batch of b sequences at a time, n is the maximum length of these sequences, and f is the dimensionality of each item in a sequence. Also, PyTorch manages the iteration for us: we just call $f_\theta(x_0 x_1 \cdots x_{n-1}, v_0)$ and it doesn't just return $(p_1, v_1) = f_\theta(x_0, v_0)$ it returns $((p_1, \dots, p_n), v_n)$.

```
url = "https://www.cl.cam.ac.uk/teaching/2021/DataSci/data/english_names.txt"
names = pandas.read_table(url, header=None, names=['X']).X.str.lower()
alphabet = list(set(''.join(names)))
alphabet_n = {c:i+1 for i,c in enumerate(alphabet)}
A = len(alphabet)+1

# Convert from strings to integer-coding (with 0 at beginning and end) then one-hot coding
names_n = [[0] + [alphabet_n[x] for x in name] + [0] for name in names]
names_n = [torch.tensor(n) for n in names_n]
names_oh = [nn.functional.one_hot(n, num_classes=A).float() for n in names_n]

class RSeq(nn.Module):
    def __init__(self, H=50, L=2):
        super().__init__()
        # Use rnn for the iterative part, producing a vector of dim. H per timestep,
        # then apply an additional map before turning it into a probability distribution of dim. A
        self.rnn = nn.GRU(input_size=A, hidden_size=H, num_layers=L, dropout=.05)
        self.map = nn.Linear(self.rnn.hidden_size, A)

    def f(self, x, v=None):  # x.shape m × b × A, v.shape L × b × H
        assert x.shape[1]==1, "This code only works with batch_size=1"
        y,v = self.rnn(x, v) # y.shape m × b × H, v.shape L × b × H
        z = self.map(y)      # z.shape m × b × A
        logp = nn.functional.log_softmax(z, dim=2) # logp.shape m × b × A
        return logp,v

    def forward(self, x): # x.shape = (n+1)*b*A, x assumed to have ∅ and □
        logp,_ = self.f(x[:-1])       # logp.shape n × b × A
        return torch.sum(logp * x[1:], dim=(0,2)) # shape b

    def generate(self):
        is_training = self.training
        self.train(False)
        with torch.no_grad():
            res = ''
            x_n,v = 0,None
            while True:
                x_n = torch.tensor(x_n).reshape(1,1)
                x_oh = nn.functional.one_hot(x_n, num_classes=A).float()
                logp,v = model.f(x_oh, v)
                p = torch.exp(logp[0,0])
                x_n = np.random.choice(A, p=p.detach().numpy())
                if x_n == 0: break
                res += alphabet[x_n-1]
        self.train(is_training)
        return res

model = RSeq()
model.train(True)
optimizer = optim.Adam(model.parameters())

with Interruptable() as check_interrupted:
    while True:
        for j in np.random.permutation(len(names_oh)):
            check_interrupted()
            optimizer.zero_grad()
            e = - model(names_oh[j][:,None,:])
            e.backward()
            optimizer.step()
```

* ✻ *

Recurrent neural networks easily be used for supervised learning tasks. Suppose for example we want to train a neural network to write captions for images. Let the the image be x, model the caption as a random sequence $Y = Y_1 Y_2 \cdots Y_n$, and fit a model with an extra neural network g_θ for processing the image and converting it to an initial state s_0.



The Transformer[63] architecture, a neural network that generates text much more impressively than does a recurrent neural network, also uses sequential generation with non-hidden state, but with an even simpler design. It generates each item in the sequence one by one, using the model $X_n \sim \text{Cat}(f_\theta(x_1 x_2 \cdots x_{n-1}))$. The cleverness consists in designing a neural network that can accept an arbitrary length sequence as its input. But from the point of view of random variables, there's barely any difference between a recurrent neural network and a transformer.

[63] Ashish Vaswani et al. "Attention Is All You Need". In: *NIPS*. 2017. URL: `http://arxiv.org/abs/1706.03762`.