```
#hide
! [ -e /content ] && pip install -Uqq fastbook
import fastbook
fastbook.setup_book()
```

```
#hide
from fastbook import *
```

A Language Model from Scratch

The Data

```
from fastai.text.all import *
path = untar_data(URLs.HUMAN_NUMBERS)
```

108.32% [32768/30252 00:00<00:00]

```
#hide
Path.BASE_PATH = path
```

```
path.ls()
```

```
(#2) [Path('train.txt'),Path('valid.txt')]
```

```
lines = L()
with open(path/'train.txt') as f: lines += L(*f.readlines())
with open(path/'valid.txt') as f: lines += L(*f.readlines())
lines
```

```
(#9998) ['one \n','two \n','three \n','four \n','five \n','six \n','seven \n','eight \n','nine \n','ten \n'...]
```

```
text = ' . '.join([l.strip() for l in lines])
text[:100]
```

```
'one . two . three . four . five . six . seven . eight . nine . ten . eleven . tw
elve . thirteen . fo'
```

```
tokens = text.split(' ')
tokens[:10]
```

```
['one', '.', 'two', '.', 'three', '.', 'four', '.', 'five', '.']
```

```
vocab = L(*tokens).unique()
vocab
```

```
(#30) ['one','.','two','three','four','five','six','seven','eight','nine'...]
```

```
word2idx = {w:i for i,w in enumerate(vocab)}
nums = L(word2idx[i] for i in tokens)
nums
```

```
(#63095) [0,1,2,1,3,1,4,1,5,1...]
```

Our First Language Model from Scratch

```
L((tokens[i:i+3], tokens[i+3]) for i in range(0,len(tokens)-4,3))
```

```
(#21031) [(['one', '.', 'two'], '.'),(['.', 'three', '.'], 'four'),(['four', '.', 'five'], '.'),(['.', 'six', '.'], 'seven'),
(['seven', '.', 'eight'], '.'),(['.', 'nine', '.'], 'ten'),(['ten', '.', 'eleven'], '.'),(['.', 'twelve', '.'], 'thirteen'),
(['thirteen', '.', 'fourteen'], '.'),(['.', 'fifteen', '.'], 'sixteen')...]
```

```
seqs = L((tensor(nums[i:i+3]), nums[i+3]) for i in range(0,len(nums)-4,3))
seqs
```

```
(#21031) [(tensor([0, 1, 2]), 1),(tensor([1, 3, 1]), 4),(tensor([4, 1, 5]), 1),(tensor([1, 6, 1]), 7),(tensor([7, 1, 8]), 1),
(tensor([1, 9, 1]), 10),(tensor([10,  1, 11]), 1),(tensor([ 1, 12,  1]), 13),(tensor([13,  1, 14]), 1),(tensor([ 1, 15,  1]),
16)...]
```

```
bs = 64
cut = int(len(seqs) * 0.8)
dls = DataLoaders.from_dsets(seqs[:cut], seqs[cut:], bs=64, shuffle=False)
```

Our Language Model in PyTorch

```
class LMModel1(Module):
    def __init__(self, vocab_sz, n_hidden):
        self.i_h = nn.Embedding(vocab_sz, n_hidden)
        self.h_h = nn.Linear(n_hidden, n_hidden)
        self.h_o = nn.Linear(n_hidden,vocab_sz)

    def forward(self, x):
        h = F.relu(self.h_h(self.i_h(x[:,0])))
        h = h + self.i_h(x[:,1])
        h = F.relu(self.h_h(h))
        h = h + self.i_h(x[:,2])
        h = F.relu(self.h_h(h))
        return self.h_o(h)
```

```
learn = Learner(dls, LMModel1(len(vocab), 64), loss_func=F.cross_entropy,
                metrics=accuracy)
learn.fit_one_cycle(4, 1e-3)
```

epoch	train_loss	valid_loss	accuracy	time
0	1.824297	1.970941	0.467554	00:02
1	1.386973	1.823242	0.467554	00:02
2	1.417556	1.654498	0.494414	00:03
3	1.376440	1.650849	0.494414	00:03

```
n,counts = 0,torch.zeros(len(vocab))
for x,y in dls.valid:
    n += y.shape[0]
    for i in range_of(vocab): counts[i] += (y==i).long().sum()
idx = torch.argmax(counts)
idx, vocab[idx.item()], counts[idx].item()/n
```

```
(tensor(29), 'thousand', 0.15165200855716662)
```

Our First Recurrent Neural Network

```
class LMModel2(Module):
    def __init__(self, vocab_sz, n_hidden):
        self.i_h = nn.Embedding(vocab_sz, n_hidden)
        self.h_h = nn.Linear(n_hidden, n_hidden)
        self.h_o = nn.Linear(n_hidden,vocab_sz)

    def forward(self, x):
        h = 0
        for i in range(3):
            h = h + self.i_h(x[:,i])
            h = F.relu(self.h_h(h))
        return self.h_o(h)
```

```
learn = Learner(dls, LMModel2(len(vocab), 64), loss_func=F.cross_entropy,
                metrics=accuracy)
learn.fit_one_cycle(4, 1e-3)
```

epoch	train_loss	valid_loss	accuracy	time
0	1.816274	1.964143	0.460185	00:03
1	1.423805	1.739964	0.473259	00:04
2	1.430327	1.685172	0.485382	00:01
3	1.388390	1.657033	0.470406	00:01

Improving the RNN

Maintaining the State of an RNN

```
class LMModel3(Module):
    def __init__(self, vocab_sz, n_hidden):
        self.i_h = nn.Embedding(vocab_sz, n_hidden)
        self.h_h = nn.Linear(n_hidden, n_hidden)
        self.h_o = nn.Linear(n_hidden,vocab_sz)
        self.h = 0

    def forward(self, x):
        for i in range(3):
            self.h = self.h + self.i_h(x[:,i])
            self.h = F.relu(self.h_h(self.h))
        out = self.h_o(self.h)
        self.h = self.h.detach()
        return out

    def reset(self): self.h = 0
```

```
m = len(seqs)//bs
m,bs,len(seqs)
```

```
(328, 64, 21031)
```

```
def group_chunks(ds, bs):
    m = len(ds) // bs
    new_ds = L()
    for i in range(m): new_ds += L(ds[i + m*j] for j in range(bs))
    return new_ds
```

```
cut = int(len(seqs) * 0.8)
dls = DataLoaders.from_dsets(
    group_chunks(seqs[:cut], bs),
    group_chunks(seqs[cut:], bs),
    bs=bs, drop_last=True, shuffle=False)
```

```
learn = Learner(dls, LMModel3(len(vocab), 64), loss_func=F.cross_entropy,
                metrics=accuracy, cbs=ModelResetter)
learn.fit_one_cycle(10, 3e-3)
```

epoch	train_loss	valid_loss	accuracy	time
0	1.677074	1.827367	0.467548	00:01
1	1.282722	1.870913	0.388942	00:01
2	1.090705	1.651794	0.462500	00:01
3	1.005215	1.615990	0.515144	00:01
4	0.963020	1.605894	0.551202	00:02
5	0.926171	1.721725	0.543750	00:02
6	0.908232	1.668949	0.555529	00:01
7	0.843980	1.725772	0.570913	00:01
8	0.811898	1.740454	0.587260	00:01
9	0.797176	1.705923	0.589423	00:01

Creating More Signal

```
sl = 16
seqs = L((tensor(nums[i:i+sl]), tensor(nums[i+1:i+sl+1]))
         for i in range(0,len(nums)-sl-1,sl))
cut = int(len(seqs) * 0.8)
dls = DataLoaders.from_dsets(group_chunks(seqs[:cut], bs),
                             group_chunks(seqs[cut:], bs),
                             bs=bs, drop_last=True, shuffle=False)
```

```
[L(vocab[o] for o in s) for s in seqs[0]]
```

```
[(#16) ['one','.','two','.','three','.','four','.','five','.'...],
 (#16) ['.','two','.','three','.','four','.','five','.','six'...]]
```

```
class LMModel4(Module):
    def __init__(self, vocab_sz, n_hidden):
        self.i_h = nn.Embedding(vocab_sz, n_hidden)
        self.h_h = nn.Linear(n_hidden, n_hidden)
        self.h_o = nn.Linear(n_hidden,vocab_sz)
        self.h = 0

    def forward(self, x):
        outs = []
        for i in range(sl):
            self.h = self.h + self.i_h(x[:,i])
            self.h = F.relu(self.h_h(self.h))
            outs.append(self.h_o(self.h))
        self.h = self.h.detach()
        return torch.stack(outs, dim=1)

    def reset(self): self.h = 0
```

```
def loss_func(inp, targ):
    return F.cross_entropy(inp.view(-1, len(vocab)), targ.view(-1))
```

```
learn = Learner(dls, LMModel4(len(vocab), 64), loss_func=loss_func,
                metrics=accuracy, cbs=ModelResetter)
learn.fit_one_cycle(15, 3e-3)
```

epoch	train_loss	valid_loss	accuracy	time
0	3.285931	3.072032	0.212565	00:00
1	2.330371	1.969522	0.425781	00:00
2	1.742316	1.841378	0.441488	00:00
3	1.470119	1.810857	0.494303	00:00
4	1.298154	1.866849	0.479248	00:00
5	1.178096	1.730558	0.529216	00:01
6	1.071567	1.744563	0.518229	00:01
7	0.980908	1.767464	0.534749	00:00
8	0.896100	1.705250	0.577881	00:01
9	0.837038	1.643039	0.591553	00:01
10	0.790128	1.673707	0.599691	00:00
11	0.745496	1.706304	0.588298	00:00
12	0.712986	1.767930	0.591471	00:00
13	0.693416	1.782771	0.582764	00:00
14	0.681424	1.722705	0.606364	00:00

Multilayer RNNs

The Model

```
class LMModel5(Module):
    def __init__(self, vocab_sz, n_hidden, n_layers):
        self.i_h = nn.Embedding(vocab_sz, n_hidden)
        self.rnn = nn.RNN(n_hidden, n_hidden, n_layers, batch_first=True)
        self.h_o = nn.Linear(n_hidden, vocab_sz)
        self.h = torch.zeros(n_layers, bs, n_hidden)

    def forward(self, x):
        res,h = self.rnn(self.i_h(x), self.h)
        self.h = h.detach()
        return self.h_o(res)

    def reset(self): self.h.zero_()
```

```
learn = Learner(dls, LMModel5(len(vocab), 64, 2),
                loss_func=CrossEntropyLossFlat(),
                metrics=accuracy, cbs=ModelResetter)
learn.fit_one_cycle(15, 3e-3)
```

epoch	train_loss	valid_loss	accuracy	time
0	3.041790	2.548715	0.455811	00:00
1	2.128514	1.708763	0.471029	00:00
2	1.699163	1.866050	0.340576	00:00
3	1.499681	1.738478	0.471517	00:00
4	1.339090	1.729537	0.494792	00:00
5	1.206317	1.835867	0.502848	00:00
6	1.088238	1.845533	0.520101	00:00
7	0.982785	1.856221	0.522624	00:00
8	0.890787	1.940329	0.525716	00:01
9	0.809582	2.028808	0.529704	00:01
10	0.743080	2.074588	0.535075	00:01
11	0.694123	2.153387	0.540039	00:01
12	0.660757	2.137583	0.547689	00:01
13	0.640675	2.169322	0.547363	00:00
14	0.630329	2.168171	0.548828	00:00

Exploding or Disappearing Activations

LSTM

Building an LSTM from Scratch

```
class LSTMCell(Module):
    def __init__(self, ni, nh):
        self.forget_gate = nn.Linear(ni + nh, nh)
        self.input_gate  = nn.Linear(ni + nh, nh)
        self.cell_gate   = nn.Linear(ni + nh, nh)
        self.output_gate = nn.Linear(ni + nh, nh)

    def forward(self, input, state):
        h,c = state
        h = torch.cat([h, input], dim=1)
        forget = torch.sigmoid(self.forget_gate(h))
        c = c * forget
        inp = torch.sigmoid(self.input_gate(h))
        cell = torch.tanh(self.cell_gate(h))
        c = c + inp * cell
        out = torch.sigmoid(self.output_gate(h))
        h = out * torch.tanh(c)
        return h, (h,c)
```

```
class LSTMCell(Module):
    def __init__(self, ni, nh):
        self.ih = nn.Linear(ni,4*nh)
        self.hh = nn.Linear(nh,4*nh)

    def forward(self, input, state):
        h,c = state
        # One big multiplication for all the gates is better than 4 smaller ones
        gates = (self.ih(input) + self.hh(h)).chunk(4, 1)
        ingate,forgetgate,outgate = map(torch.sigmoid, gates[:3])
        cellgate = gates[3].tanh()

        c = (forgetgate*c) + (ingate*cellgate)
        h = outgate * c.tanh()
        return h, (h,c)
```

```
t = torch.arange(0,10); t
```

```
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
t.chunk(2)
```

```
(tensor([0, 1, 2, 3, 4]), tensor([5, 6, 7, 8, 9]))
```

Training a Language Model Using LSTMs

```
class LMModel6(Module):
    def __init__(self, vocab_sz, n_hidden, n_layers):
        self.i_h = nn.Embedding(vocab_sz, n_hidden)
        self.rnn = nn.LSTM(n_hidden, n_hidden, n_layers, batch_first=True)
        self.h_o = nn.Linear(n_hidden, vocab_sz)
        self.h = [torch.zeros(n_layers, bs, n_hidden) for _ in range(2)]

    def forward(self, x):
        res,h = self.rnn(self.i_h(x), self.h)
        self.h = [h_.detach() for h_ in h]
        return self.h_o(res)

    def reset(self):
        for h in self.h: h.zero_()
```

```
learn = Learner(dls, LMModel6(len(vocab), 64, 2),
                loss_func=CrossEntropyLossFlat(),
                metrics=accuracy, cbs=ModelResetter)
learn.fit_one_cycle(15, 1e-2)
```

epoch	train_loss	valid_loss	accuracy	time
0	3.026114	2.772101	0.153076	00:01
1	2.216185	2.089064	0.269124	00:01
2	1.613936	1.826342	0.478678	00:01
3	1.317173	2.115477	0.492594	00:01
4	1.084439	1.966246	0.607422	00:01
5	0.853946	1.723211	0.589844	00:01
6	0.609810	1.802983	0.629639	00:01
7	0.424614	1.669696	0.676188	00:01
8	0.274180	1.764627	0.687174	00:01
9	0.185413	1.528294	0.717855	00:01
10	0.119011	1.594475	0.719238	00:01
11	0.079207	1.731244	0.711751	00:01
12	0.056416	1.748391	0.719727	00:01
13	0.044439	1.725030	0.724772	00:01
14	0.038967	1.745424	0.720784	00:01

Regularizing an LSTM

Dropout

```
class Dropout(Module):
    def __init__(self, p): self.p = p
    def forward(self, x):
        if not self.training: return x
        mask = x.new(*x.shape).bernoulli_(1-p)
        return x * mask.div_(1-p)
```

Activation Regularization and Temporal Activation Regularization

Training a Weight-Tied Regularized LSTM

```
class LMModel7(Module):
    def __init__(self, vocab_sz, n_hidden, n_layers, p):
        self.i_h = nn.Embedding(vocab_sz, n_hidden)
        self.rnn = nn.LSTM(n_hidden, n_hidden, n_layers, batch_first=True)
        self.drop = nn.Dropout(p)
        self.h_o = nn.Linear(n_hidden, vocab_sz)
```

```
        self.h_o.weight = self.i_h.weight
        self.h = [torch.zeros(n_layers, bs, n_hidden) for _ in range(2)]

    def forward(self, x):
        raw,h = self.rnn(self.i_h(x), self.h)
        out = self.drop(raw)
        self.h = [h_.detach() for h_ in h]
        return self.h_o(out),raw,out

    def reset(self):
        for h in self.h: h.zero_()
```

```
learn = Learner(dls, LMModel7(len(vocab), 64, 2, 0.5),
                loss_func=CrossEntropyLossFlat(), metrics=accuracy,
                cbs=[ModelResetter, RNNRegularizer(alpha=2, beta=1)])
```

```
learn = TextLearner(dls, LMModel7(len(vocab), 64, 2, 0.4),
                    loss_func=CrossEntropyLossFlat(), metrics=accuracy)
```

```
learn.fit_one_cycle(15, 1e-2, wd=0.1)
```

epoch	train_loss	valid_loss	accuracy	time
0	2.461772	1.946499	0.509277	00:01
1	1.516596	1.259043	0.637370	00:01
2	0.801911	0.857456	0.779460	00:01
3	0.403066	0.793918	0.790853	00:01
4	0.210355	0.758285	0.823324	00:01
5	0.115218	0.806547	0.826742	00:01
6	0.072559	0.742029	0.831299	00:01
7	0.049400	0.749732	0.846191	00:01
8	0.034875	0.681907	0.846354	00:01
9	0.027917	0.612487	0.870361	00:01
10	0.023385	0.806724	0.830078	00:01
11	0.020127	0.751685	0.841064	00:01
12	0.016826	0.785721	0.834798	00:01
13	0.014216	0.773120	0.837484	00:01
14	0.012662	0.796994	0.836019	00:01

Conclusion

Questionnaire

1. If the dataset for your project is so big and complicated that working with it takes a significant amount of time, what should you do?
2. Why do we concatenate the documents in our dataset before creating a language model?
3. To use a standard fully connected network to predict the fourth word given the previous three words, what two tweaks do we need to make to our model?
4. How can we share a weight matrix across multiple layers in PyTorch?
5. Write a module that predicts the third word given the previous two words of a sentence, without peeking.
6. What is a recurrent neural network?
7. What is "hidden state"?
8. What is the equivalent of hidden state in `LMModel1`?
9. To maintain the state in an RNN, why is it important to pass the text to the model in order?
10. What is an "unrolled" representation of an RNN?
11. Why can maintaining the hidden state in an RNN lead to memory and performance problems? How do we fix this problem?
12. What is "BPTT"?
13. Write code to print out the first few batches of the validation set, including converting the token IDs back into English strings, as we showed for batches of IMDb data in <>.
14. What does the `ModelResetter` callback do? Why do we need it?
15. What are the downsides of predicting just one output word for each three input words?

16. Why do we need a custom loss function for `LMModel4`?
17. Why is the training of `LMModel4` unstable?
18. In the unrolled representation, we can see that a recurrent neural network actually has many layers. So why do we need to stack RNNs to get better results?
19. Draw a representation of a stacked (multilayer) RNN.
20. Why should we get better results in an RNN if we call `detach` less often? Why might this not happen in practice with a simple RNN?
21. Why can a deep network result in very large or very small activations? Why does this matter?
22. In a computer's floating-point representation of numbers, which numbers are the most precise?
23. Why do vanishing gradients prevent training?
24. Why does it help to have two hidden states in the LSTM architecture? What is the purpose of each one?
25. What are these two states called in an LSTM?
26. What is tanh, and how is it related to sigmoid?
27. What is the purpose of this code in `LSTMCell`: `h = torch.cat([h, input], dim=1)`
28. What does `chunk` do in PyTorch?
29. Study the refactored version of `LSTMCell` carefully to ensure you understand how and why it does the same thing as the non-refactored version.
30. Why can we use a higher learning rate for `LMModel6`?
31. What are the three regularization techniques used in an AWD-LSTM model?
32. What is "dropout"?
33. Why do we scale the acitvations with dropout? Is this applied during training, inference, or both?
34. What is the purpose of this line from `Dropout`: `if not self.training: return x`
35. Experiment with `bernoulli_` to understand how it works.
36. How do you set your model in training mode in PyTorch? In evaluation mode?
37. Write the equation for activation regularization (in math or code, as you prefer). How is it different from weight decay?
38. Write the equation for temporal activation regularization (in math or code, as you prefer). Why wouldn't we use this for computer vision problems?
39. What is "weight tying" in a language model?

Further Research

1. In `LMModel2`, why can `forward` start with `h=0`? Why don't we need to say `h=torch.zeros(...)`?
2. Write the code for an LSTM from scratch (you may refer to <>).
3. Search the internet for the GRU architecture and implement it from scratch, and try training a model. See if you can get results similar to those we saw in this chapter. Compare your results to the results of PyTorch's built in `GRU` module.
4. Take a look at the source code for AWD-LSTM in fastai, and try to map each of the lines of code to the concepts shown in this chapter.