Chapter 4

Feedforward Neural Networks

4.1 Motivation

Let's start with our logistic regression model from before:

$$P(k \mid d) = \underset{k'=k}{\text{softmax}}\left(\lambda(k') + \sum_{w \in d} \lambda(k', w)\right). \tag{4.1}$$

Recall that this model gives us a lot of flexibility to add new features. For example, we might want a feature for words end with the suffix -ing. Or for words that are related to technology. Designing these features well is known as "feature engineering." Some people like feature engineering because it's an opportunity to bring in your knowledge about natural language and your task. Some people don't like feature engineering because it takes time, and they'd rather have the computer do it for them – much like we saw how a topic model can automatically discover topics. Automating feature engineering is what neural networks claim to offer.

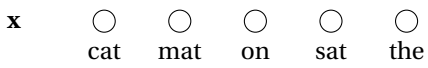
4.2 Notation

Since this chapter is about neural networks, we need to rewrite logistic regression using the notation and terminology of neural networks. Let V be the vocabulary and let K be the set of classes. For concreteness, suppose that the vocabulary is $V = \{\text{cat}, \text{mat}, \text{on}, \text{sat}, \text{the}\}$, and the classes are fiction genres, $K = \{\text{fantasy}, \text{horror}, \text{mystery}, \text{sci-fi}\}$.

Assume that all the word types in V are numbered (consecutively from 1) and all the classes in K are numbered (consecutively from 1). Let $\mathbf{x}$ be the $|V| \times 1$ vector of word counts in a document. For example, if the vocabulary is $V = \{\text{cat}, \text{mat}, \text{on}, \text{sat}, \text{the}\}$, then the sentence "the cat sat on the mat" would be encoded as

$$\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 2 \end{bmatrix}.$$

Pictorially, **x** is drawn as a row of nodes (one for each word type), each of which has a value (the count of the word). In neural network terminology, it's a *layer* of *units*, each of which has an *activation*.



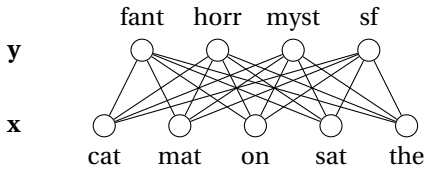
Now let **W** be a $|K| \times |V|$ matrix, and let **b** be a $|K| \times 1$ vector. These are equivalent to the λ's in (4.1), as follows:

$$W_{kw} = \lambda(k, w)$$
$$b_k = \lambda(k).$$

I apologize for the change in notation, but the payoff is that now the model can be written very compactly as:

$$\mathbf{y} = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b}).$$

Pictorially, we draw **y**, the vector of class probabilities, as another layer on top of **x**. (Inputs are always down and outputs are always up.)

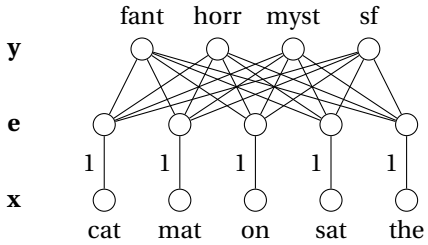


Each connection transmits the activation of the lower unit to the upper unit, multiplying it by a weight, which is adjusted during training. The upper unit, in turn, may perform some further function, called an *activation function*, to the weighted sum it receives from its input connections. In this case, the activation function is a softmax. (The picture above shows the connections whose weights are **W**, but the connections whose weights are **b** (known as *biases*) are rarely, if ever, drawn.)

4.3 Word Embeddings

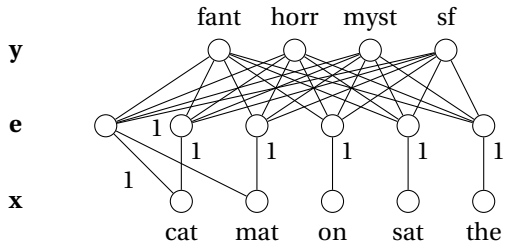
Now we want to make a "universal feature" that can be configured to behave like any feature we might dream of, with the hope that we can simply give the computer (say) 100 universal features and have it figure out automatically how to configure them.

We make a small change to our network:



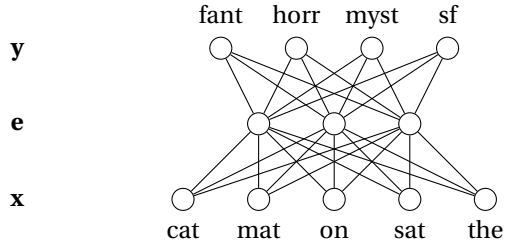
We added a new layer **e**, and the connections from **x** to **e** all have weights fixed to 1, as shown. Atypically, **e** doesn't apply a softmax or any other activation function.

Now suppose that we want to add a new feature that counts the number of nouns; assume that we know the part-of-speech of each word in V. That's easy to do in this graphical representation:



We added a new feature unit, with incoming connections from the nouns (this defines the feature) and outgoing connections to the output units (these are the feature weights). So far, this is nothing new. This is still plain old logistic regression; only the notation has changed.

But now, to get a universal feature, all we have to do is make a feature unit whose incoming weights are not fixed. Instead, the incoming weights are learned along with the rest of the weights. Here's what the network looks like with three such units:



Switching back to equations, our model is now

$$\begin{aligned} \mathbf{e} &= \mathbf{E}\mathbf{x} \\ \mathbf{y} &= \text{softmax}(\mathbf{W}\mathbf{e} + \mathbf{b}). \end{aligned} \tag{4.2}$$

The addition of a new layer is very similar indeed to the addition of topics to the naive Bayes classifier from the last chapter. But **e** is not a vector of probabilities; it's just a vector of numbers.

The matrix **E** has one column for each word type in the vocabulary. That is, for each word w, $\mathbf{E}_{:,w}$ is a vector that can be thought of as a representation of w, known as a *word embedding*. Word embeddings have received an enormous amount of attention in the last few years, because they seem to capture a sometimes remarkable amount of information about word meanings.

4.4 More Layers

Another traditional way of cooking up new features is to form conjunctions and disjunctions of features. If I already have a feature for nouns and a feature for technology-related words, then I can form a new feature for words that are both nouns *and* technology-related, as well as (for whatever reason) a feature

for words that are either nouns *or* technology-related. In principle, you could make conjunctions and disjunctions of three, four, or more features. The downside is that this causes a huge blowup in the number of features. It would be nice to automatically figure out which combinations are the good ones – and we can, by creating a "universal feature combiner."

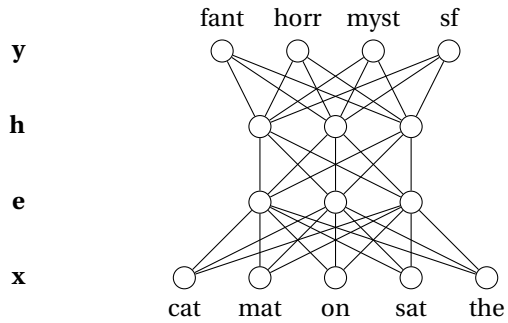
Define the function

$$H(x) = \begin{cases} 0 & \text{if } x < 0 \\ \frac{1}{2} & \text{if } x = 0 \\ 1 & \text{if } x > 0. \end{cases}$$

Then we can define *and* and *or* as follows:

$$x \wedge y = H(x + y - 1.5)$$
$$x \vee y = H(x + y - 0.5)$$

Both combination operations are just H applied to a linear combination of the arguments (plus a bias), and this remains true for any number of arguments. So just as we made "universal features," we can make a layer of "universal feature combiners," **h**:



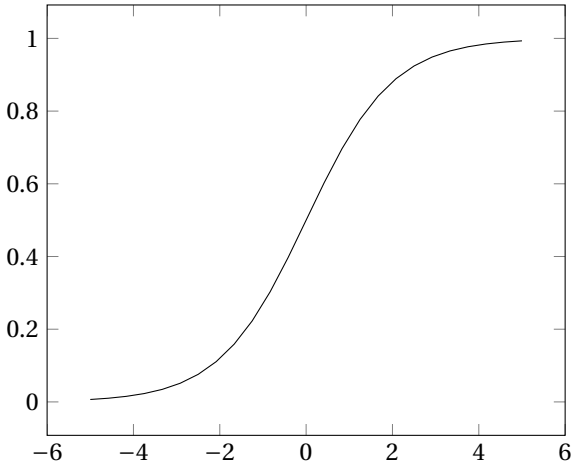
In equations,

$$\mathbf{e} = \mathbf{E}\mathbf{x}$$
$$\mathbf{h} = H(\mathbf{V}\mathbf{e} + \mathbf{a})$$
$$\mathbf{y} = \text{softmax}(\mathbf{W}\mathbf{h} + \mathbf{b}).$$

By learning the connection weights **V** and **a**, we can get all kinds of combinations of the features **e**. The only problem is that H is not differentiable at zero. (Also, it's flat everywhere else, which makes it hard to optimize.) So we replace it with the *sigmoid* function

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)},$$

which looks like this:



which clearly is differentiable (its derivative is $\text{sigmoid}(x)(1-\text{sigmoid}(x))$).

So we get the following model, which is known as a *feedforward neural network* or *multilayer perceptron*:

$$\begin{aligned}\mathbf{e} &= \mathbf{E}\mathbf{x}\\ \mathbf{h} &= \text{sigmoid}(\mathbf{V}\mathbf{e}+\mathbf{a})\\ \mathbf{y} &= \text{softmax}(\mathbf{W}\mathbf{h}+\mathbf{b}).\end{aligned}$$

If we wanted to, we could insert even more hidden layers – the "deep" in "deep learning" refers to number of hidden layers in a neural network.

In practice, tanh is usually used in place of sigmoid. The tanh function is just a scaled and shifted version of sigmoid ($\tanh(x) = 2\,\text{sigmoid}(2x)-1$), and turns out to make the network easier to train.

4.5 Training

The usual training method for neural networks is called *backpropagation*, which is really just stochastic gradient ascent (or descent), just as we've been training our other models. The gradients for these models can get quite complicated. Automatic differentiation is your friend, but it's good to have a basic understanding of how it happens.

4.5.1 Backpropagation

Using the model above, we want to compute the partial derivatives of $L = \log y_k$ (where k is the correct class) with respect to $\mathbf{E}$, $\mathbf{V}$, $\mathbf{a}$, $\mathbf{W}$, and $\mathbf{b}$. Let's introduce some new variables to make this easier:

$$\mathbf{e} = \mathbf{E}\mathbf{x} \tag{4.3}$$

$$\tilde{\mathbf{h}} = \mathbf{V}\mathbf{e}+\mathbf{a} \tag{4.4}$$

$$\mathbf{h} = \text{sigmoid}(\tilde{\mathbf{h}}) \tag{4.5}$$

$$\tilde{\mathbf{y}} = \mathbf{W}\mathbf{h}+\mathbf{b} \tag{4.6}$$

$$\mathbf{y} = \text{softmax}(\tilde{\mathbf{y}}). \tag{4.7}$$

We use the chain rule for partial derivatives to compute the partial derivative of L with respect to each of the variables, starting from the end and working our way backwards. Where the resulting equations look simpler in terms of vectors, we show them in the right column.

$$\frac{\partial L}{\partial y_k} = \frac{1}{y_k}$$
$$\frac{\partial L}{\partial y_i} = 0 \qquad (i \neq k)$$

From (4.7):

$$\begin{aligned}\frac{\partial L}{\partial \tilde{y}_i} &= \frac{\partial L}{\partial y_k}\frac{\partial y_k}{\partial \tilde{y}_i} \\ &= \frac{1}{y_k} y_k(\delta(i,k) - y_i) \\ &= \delta(i,k) - y_i\end{aligned}$$

From (4.6):

$$\begin{aligned}\frac{\partial L}{\partial W_{ij}} &= \frac{\partial L}{\partial \tilde{y}_i}\frac{\partial \tilde{y}_i}{\partial W_{ij}} \\ &= \frac{\partial L}{\partial \tilde{y}_i} \cdot h_j && \frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \tilde{\mathbf{y}}} \otimes \mathbf{h} \\ \frac{\partial L}{\partial b_i} &= \frac{\partial L}{\partial \tilde{y}_i}\frac{\partial \tilde{y}_i}{\partial b_i} \\ &= \frac{\partial L}{\partial \tilde{y}_i} \cdot 1 && \frac{\partial L}{\partial \mathbf{b}} = \frac{\partial L}{\partial \tilde{\mathbf{y}}} \\ \frac{\partial L}{\partial h_j} &= \sum_i \frac{\partial L}{\partial \tilde{y}_i}\frac{\partial \tilde{y}_i}{\partial h_j} \\ &= \sum_i \frac{\partial L}{\partial \tilde{y}_i} \cdot W_{ij} && \frac{\partial L}{\partial \mathbf{h}} = \frac{\partial L}{\partial \tilde{\mathbf{y}}}\mathbf{W}\end{aligned}$$

And so on. Each unit receives a gradient from the units connected to it above and forms their weighted sum. And each unit sends its gradient to all the units connected to it below. Since, in general, a gradient might get used multiple times, it's important to memoize (cache) them and not recompute them over and over. This is an example of dynamic programming.

4.5.2 Practical details

Because the objective function can have local minima, initialization is very important. You definitely can't initialize all the parameters to zero, because the units in a hidden layer are all identical and won't be able to learn to do different things. Initializing them randomly and uniformly from $[-0.01, +0.01]$ is not too bad. Fancier schemes have been proposed by Glorot and Bengio (2010) and Saxe, McClelland, and Ganguli (2014).

Choosing the learning rate is also important; try 0.1 but experiment with bigger and smaller powers of ten. You can also try decaying the learning rate; for example, whenever the accuracy on the dev set gets worse, halve the learning rate.

In general with stochastic gradient ascent, but especially, it seems, with neural networks, different learning rates may be appropriate for different parameters. A learning rate of 0.1 might be too slow for some parameters but way too fast for others. There are a lot of tools for fixing this problem. The bluntest is *gradient clipping*: if a parameter's partial derivative has an absolute value bigger than some upper bound (usually 5), then set it to the upper bound. More principled, but still fairly simple, is Adagrad (Duchi, Hazan, and Singer, 2011), which automatically chooses a different learning rate for each feature.

4.6 Experiment

We ran the network described above on the gender-classification task, using

- the top 10000 most frequent word types (with others replaced by `<unk>`), mainly for speed but also to reduce overfitting
- 20 units for both **e** and **h**
- Adagrad instead of stochastic gradient ascent
- learning rate of 0.01
- just one iteration (Adagrad speeds up convergence quite a bit)

The results were:

naive Bayes	65.1
logistic regression	66.7
feedforward neural network	69.1

Bibliography

Duchi, John, Elad Hazan, and Yoram Singer (2011). "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *J. Machine Learning Research* 12, pp. 2121–2159.

Glorot, Xavier and Yoshua Bengio (2010). "Understanding the difficulty of training deep feedforward neural networks". In: *Proc. AISTATS*.

Saxe, Andrew M., James L. McClelland, and Surya Ganguli (2014). "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks". In: *Proc. ICLR*.