5 Language Models 3: Neural Networks and Feed-forward Language Models

In this chapter, we describe language models based on **neural networks**, a way to learn more sophisticated functions to improve the accuracy of our probability estimates with less feature engineering.

5.1 Potential and Problems with Combination Features

farmers eat	steak → **high** hay → **low**	cows eat	steak → **low** hay → **high**
farmers grow	steak → **low** hay → **high**	cows grow	steak → **low** hay → **low**

Figure 6: An example of the effect that combining multiple words can have on translation probabilities.

Before moving into the technical detail of neural networks, first let's take a look at a motivating example in Figure 6. From the example, we can see e_{t-2} = "farmers" is compatible with e_t = "hay" (in the context "farmers grow hay"), and e_{t-1} = eat is also compatible (in the context "cows eat hay"). If we are using a log-linear model with one set of features dependent on e_{t-1}, and another set of features dependent on e_{t-2}, neither set of features can rule out the unnatural phrase "farmers eat hay."

One way we can fix this problem is by creating another set of features where we learn one vector for each pair of words e_{t-2}, e_{t-1}. If this is the case, our vector for the context e_{t-2} = "farmers", e_{t-1} = "eat" could assign a low score to "hay", resolving this problem. However, adding these combination features has one major disadvantage: it greatly expands the parameters: instead of $O(|V|^2)$ parameters for each pair e_{i-1}, e_i, we need $O(|V|^3)$ parameters for each triplet e_{i-2}, e_{i-1}, e_i. These numbers greatly increase the amount of memory used by the model, and if there are not enough training examples, the parameters may not be learned properly.

Because of both the importance of and difficulty in learning using these combination features, a number of methods have been proposed to handle these features. These include **support vector machines** [10] and **neural networks** [22, 11], which we will cover in more detail below.

5.2 A Brief Overview of Neural Networks

To understand neural networks in more detail, let's take a very simple example of a function that we cannot learn with a simple linear classifier like the ones we used in the last chapter: a function that takes an input $\boldsymbol{x} \in \{-1, 1\}^2$ and outputs $y = 1$ if both x_1 and x_2 are equal and $y = -1$ otherwise. This function is shown in Figure 7.

A first attempt at solving this function might define a linear model (like the log-linear models from the previous chapter) that solves this problem with a model of the following

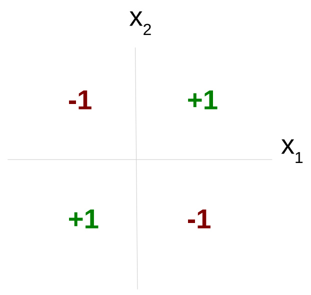

Figure 7: A function that cannot be solved by a linear transformation.

form:

$$y = W\boldsymbol{x} + b. \tag{33}$$

However, this class of functions is not powerful enough to represent the function at hand.[10]

Thus, we turn to a slightly more complicated class of functions taking the following form:

$$\begin{aligned} \boldsymbol{h} &= \text{step}(W_{xh}\boldsymbol{x} + \boldsymbol{b}_h) \\ y &= \boldsymbol{w}_{hy}\boldsymbol{h} + b_y. \end{aligned} \tag{34}$$

Computation is split into two stages: calculation of the **hidden layer**, which takes in input $\boldsymbol{x}$ and outputs a vector of hidden variables $\boldsymbol{h}$, and calculation of the **output layer**, which takes in $\boldsymbol{h}$ and calculates the final result y. Both layers consist of an **affine transform**[11] using weights W and biases $\boldsymbol{b}$, followed by a step$(\cdot)$ function, which calculates the following function:

$$\text{step}(x) = \begin{cases} 1 & \text{if } x > 0, \\ -1 & \text{otherwise.} \end{cases} \tag{35}$$

This function is one example of a class of neural networks called **multi-layer perceptrons** (MLPs). In general, MLPs consist one or more hidden layers that consist of an affine transform followed by a non-linear function (such as the step function used here), culminating in an output layer that calculates some variety of output.

Figure 8 demonstrates why this type of network does a better job of representing the non-linear function of Figure 7. In short, we can see that the first hidden layer *transforms* the input $\boldsymbol{x}$ into a hidden vector $\boldsymbol{h}$ in a different space that is more conducive for modeling our final function. Specifically in this case, we can see that $\boldsymbol{h}$ is now in a space where we can define a linear function (using $\boldsymbol{w}_y$ and b_y) that correctly calculates the desired output y.

As mentioned above, MLPs are one specific variety of neural network. More generally, neural networks can be thought of as a chain of functions (such as the affine transforms and step functions used above, but also including many, many others) that takes some input and calculates some desired output. The power of neural networks lies in the fact that chaining together a variety of simpler functions makes it possible to represent more complicated functions in an easily trainable, parameter-efficient way.[12] We will see more about training in

[10] *Question: Prove this by trying to solve the system of equations.*

[11] A fancy name for a multiplication followed by an addition.

[12] In fact, the simple single-layer MLP described above is a **universal function approximator** [15], which means that it can approximate any function to arbitrary accuracy if its hidden vector $\boldsymbol{h}$ is large enough.

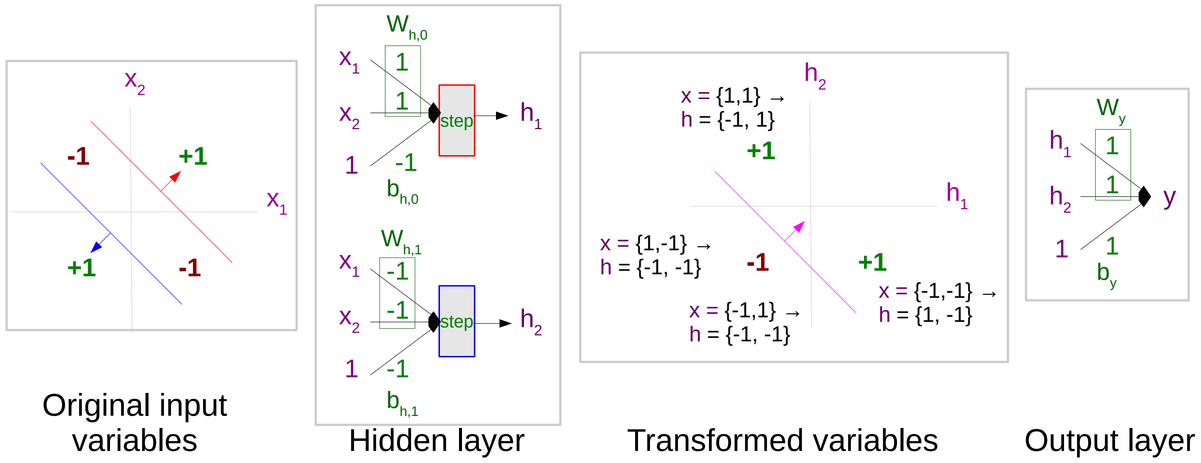

Figure 8: A simple neural network that represents the nonlinear function of Figure 7.

Section 5.3 and give some more examples of how these can be more parameter efficient in the discussion of neural network language models in Section 5.5.

5.3 Training Neural Networks

Now that we have a model in Equation 34, we would like to train its parameters W_{mh}, $\boldsymbol{b}_h$, $\boldsymbol{w}_{hy}$, and b_y. To do so, remembering our gradient-based training methods from the last chapter, we need to define the loss function $\ell(\cdot)$, calculate the derivative of the loss with respect to the parameters, then take a step in the direction that will reduce the loss. For our loss function, let's use the **squared-error loss**, a commonly used loss function for regression problems which measures the difference between the calculated value y and correct value y^* as follows

$$\ell(y^*, y) = (y^* - y)^2. \tag{36}$$

Next, we need to calculate derivatives. Here, we run into one problem: the step$(\cdot)$ function is not very derivative friendly, with its derivative being:

$$\frac{d\text{step}(x)}{dx} = \begin{cases} \text{undefined} & \text{if } x = 0, \\ 0 & \text{otherwise.} \end{cases} \tag{37}$$

Because of this, it is more common to use other non-linear functions, such as the hyperbolic tangent (tanh) function. The tanh function, as shown in Figure 9, looks very much like a softened version of the step function that has a continuous gradient everywhere, making it more conducive to training with gradient-based methods. There are a number of other alternatives as well, the most popular of which being the rectified linear unit (RelU)

$$\text{RelU}(x) = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{38}$$

shown in the left of Figure 9. In short, RelUs solve the problem that the tanh function gets "saturated" and has very small gradients when its input x is very small or very large.

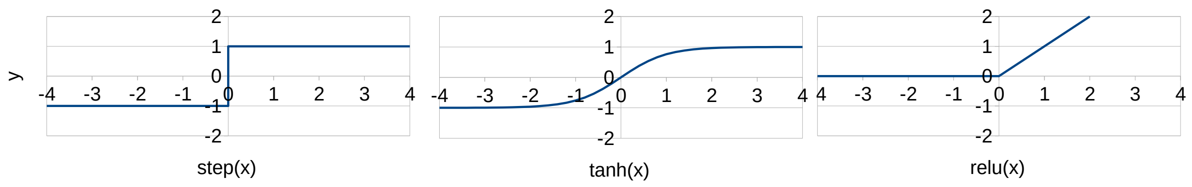

Figure 9: Types of non-linearities.

Empirical results have often shown it to be an effective alternative to tanh, including for the language modeling task described in this chapter [27].

So let's say we swap in a tanh non-linearity instead of the step function to our network, we can now proceed to calculate derivatives like we did in Section 4.3. First, we perform the full calculation of the loss function:

$$\begin{aligned} \boldsymbol{h}' &= W_{xh}\boldsymbol{x} + \boldsymbol{b}_h \\ \boldsymbol{h} &= \tanh(\boldsymbol{h}') \\ y &= \boldsymbol{w}_{hy}\boldsymbol{h} + b_y \\ \ell &= (y^* - y)^2. \end{aligned} \tag{39}$$

Then, again using the chain rule, we calculate the derivatives of each set of parameters:

$$\begin{aligned} \frac{d\ell}{db_y} &= \frac{d\ell}{dy}\frac{dy}{db_y} \\ \frac{d\ell}{d\boldsymbol{w}_{hy}} &= \frac{d\ell}{dy}\frac{dy}{d\boldsymbol{w}_{hy}} \\ \frac{d\ell}{d\boldsymbol{b}_h} &= \frac{d\ell}{dy}\frac{dy}{d\boldsymbol{h}}\frac{d\boldsymbol{h}}{d\boldsymbol{h}'}\frac{d\boldsymbol{h}'}{d\boldsymbol{b}_h} \\ \frac{d\ell}{dW_{xh}} &= \frac{d\ell}{dy}\frac{dy}{d\boldsymbol{h}}\frac{d\boldsymbol{h}}{d\boldsymbol{h}'}\frac{d\boldsymbol{h}'}{dW_{xh}}. \end{aligned} \tag{40}$$

We could go through all of the derivations above by hand and precisely calculate the gradients of all parameters in the model. Interested readers are free to do so, but even for a simple model like the one above, it is quite a lot of work and error prone. For more complicated models, like the ones introduced in the following chapters, this is even more the case.

Fortunately, when we actually implement neural networks on a computer, there is a very useful tool that saves us a large portion of this pain: **automatic differentiation** (autodiff) [29, 13]. To understand automatic differentiation, it is useful to think of our computation in Equation 39 as a data structure called a **computation graph**, two examples of which are shown in Figure 10. In these graphs, each node represents either an input to the network or the result of one computational operation, such as a multiplication, addition, tanh, or squared error. The first graph in the figure calculates the function of interest itself and would be used when we want to make predictions using our model, and the second graph calculates the loss function and would be used in training.

Automatic differentiation is a two-step dynamic programming algorithm that operates over the second graph and performs:

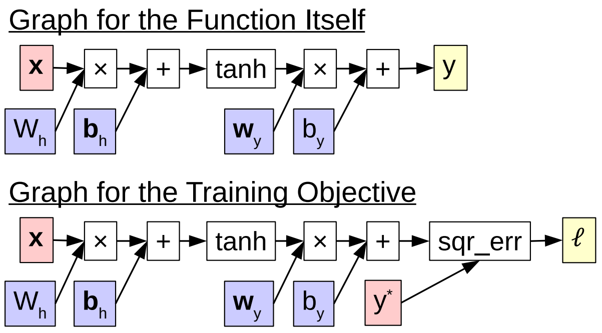
Graph for the Function Itself
x
×
+
tanh
×
+
y
W_h
b_h
w_y
b_y
Graph for the Training Objective
x
×
+
tanh
×
+
sqr_err
ℓ
W_h
b_h
w_y
b_y
y*

Figure 10: Computation graphs for the function itself, and the loss function.

- **Forward calculation**, which traverses the nodes in the graph in topological order, calculating the actual result of the computation as in Equation 39.
- **Back propagation**, which traverses the nodes in reverse topological order, calculating the gradients as in Equation 40.

The nice thing about this formulation is that while the overall function calculated by the graph can be relatively complicated, as long as it can be created by combining multiple simple nodes we are able to use automatic differentiation to calculate its derivatives using this dynamic program without doing the derivation by hand.

Thus, to implement a general purpose training algorithm for neural networks, it is necessary to implement these two dynamic programs, as well as the atomic forward function and backward derivative computations for each type of node that we would need to use. Unfortunately, this in itself is not trivial. Fortunately, there are now a plethora of toolkits that either perform general-purpose auto-differentiation [3, 14], or auto-differentiation specifically tailored for machine learning and neural networks [1, 6, 8, 24, 21]. These implement the data structures, nodes, back-propogation, and parameter optimization algorithms needed to train neural networks in an efficient and reliable way, allowing practitioners to get started with designing their models. In the following sections, we will take this approach, taking a look at how to create our models of interest in a toolkit called DyNet[13], which has a programming interface that makes it relatively easy to implement the sequence-to-sequence models covered by this course.[14]

5.4 An Example Implementation

Figure 11 shows an example of implementing the above neural network in DyNet, which we'll step through line-by-line. Lines 1-2 import the necessary libraries. Lines 4-5 specify parameters of the models: the size of the hidden vector $\boldsymbol{h}$ and the number of epochs (passes through the data) for which we'll perform training. Line 7 initializes a DyNet model, which will store all the parameters we are attempting to learn. Lines 8-11 initialize parameters W_{xh}, $\boldsymbol{b}_h$, $\boldsymbol{w}_{hy}$, and b_y to be the appropriate size so that dimensions in the equations for Equation 39 match. Line 12 initializes a "trainer", which will update the parameters in the

[13] `http://github.com/clab/dynet`

[14] It is also developed by the author of this course, so the decision might have been a wee bit biased.

```python
import dynet as dy
import random
# Parameters of the model and training
HIDDEN_SIZE = 20
NUM_EPOCHS = 20
# Define the model and SGD optimizer
model = dy.Model()
W_xh_p = model.add_parameters((HIDDEN_SIZE, 2))
b_h_p = model.add_parameters(HIDDEN_SIZE)
W_hy_p = model.add_parameters((1, HIDDEN_SIZE))
b_y_p = model.add_parameters(1)
trainer = dy.SimpleSGDTrainer(model)
# Define the training data, consisting of (x,y) tuples
data = [([1,1],1), ([-1,1],-1), ([1,-1],-1), ([-1,-1],1)]
# Define the function we would like to calculate
def calc_function(x):
  dy.renew_cg()
  w_xh = dy.parameter(w_xh_p)
  b_h = dy.parameter(b_h_p)
  W_hy = dy.parameter(W_hy_p)
  b_y = dy.parameter(b_y_p)
  x_val = dy.inputVector(x)
  h_val = dy.tanh(w_xh * x_val + b_h)
  y_val = W_hy * h_val + b_y
  return y_val
# Perform training
for epoch in range(NUM_EPOCHS):
  epoch_loss = 0
  random.shuffle(data)
  for x, ystar in data:
    y = calc_function(x)
    loss = dy.squared_distance(y, dy.scalarInput(ystar))
    epoch_loss += loss.value()
    loss.backward()
    trainer.update()
  print("Epoch %d: loss=%f" % (epoch, epoch_loss))
# Print results of prediction
for x, ystar in data:
  y = calc_function(x)
  print("%r -> %f" % (x, y.value()))
```

Figure 11: Computation graphs for the function itself, and the loss function.

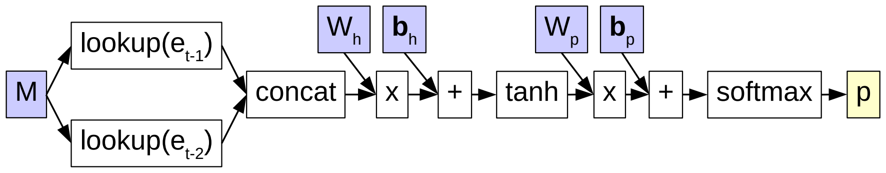

Figure 12: A computation graph for a tri-gram feed-forward neural language model.

model according to an update strategy (here we use simple stochastic gradient descent, but trainers for AdaGrad, Adam, and other strategies also exist). Line 14 creates the training data for the function in Figure 7.

Lines 16-25 define a function that takes input $\boldsymbol{x}$ and creates a computation graph to calculate Equation 39. First, line 17 creates a new computation graph to hold the computation for this particular training example. Lines 18-21 take the parameters (stored in the model) and adds them to the computation graph as DyNet variables for this particular training example. Line 22 takes a Python list representing the current input and puts it into the computation graph as a DyNet variable. Line 23 calculates the hidden vector $\boldsymbol{h}$, Line 24 calculates the value y, and Line 25 returns it.

Lines 27-36 perform training for `NUM_EPOCHS` passes over the data (one pass through the training data is usually called an "epoch"). Line 28 creates a variable to keep track of the loss for this epoch for later reporting. Line 29 shuffles the data, as recommended in Section 4.2. Lines 30-35 perform stochastic gradient descent, looping over each of the training examples. Line 31 creates a computation for the function itself, and Line 32 adds computation for the loss function. Line 33 runs the forward calculation to calculate the loss and adds it to the loss for this epoch. Line 34 runs back propagation, and Line 35 updates the model parameters. At the end of the epoch, we print the loss for the epoch in Line 36 to make sure that the loss is going down and our model is converging.

Finally, at the end of training in Lines 38-40, we print the output results. In an actual scenario, this would be done on a separate set of test data.

5.5 Neural-network Language Models

Now that we have the basics down, it is time to apply neural networks to language modeling [20, 4]. A feed-forward neural network language model is very much like the log-linear language model that we mentioned in the previous section, simply with the addition of one or more non-linear layers before the output.

First, let's recall the tri-gram log-linear language model. In this case, assume we have two sets of features expressing the identity of e_{t-1} (represented as $W^{(1)}$) and e_{t-2} (as $W^{(2)}$), the equation for the log-linear model looks like this:

$$\begin{aligned}\boldsymbol{s} &= W^{(1)}_{\cdot,e_{t-1}} + W^{(2)}_{\cdot,e_{t-2}} + \boldsymbol{b} \\ \boldsymbol{p} &= \text{softmax}(\boldsymbol{s}), \end{aligned} \tag{41}$$

where we add the appropriate columns from the weight matricies to the bias to get the score, then take the softmax to turn it into a probability.

```python
# Define the lookup parameters at model definition time
# VOCAB_SIZE is the number of words in the vocabulary
# EMBEDDINGS_SIZE is the length of the word embedding vector
M_p = model.add_lookup_parameters((VOCAB_SIZE, EMBEDDING_SIZE))
# Load the parameters into the computation graph
M = dy.lookup_parameter(M_p)
# And look up the vector for word i
m = M[i]
```

Figure 13: Code for looking things up in DyNet.

Compared to this, a tri-gram neural network model with a single layer is structured as shown in Figure 12 and described in equations below:

$$\begin{aligned}
\boldsymbol{m} &= \mathrm{concat}(M_{\cdot,e_{t-2}}, M_{\cdot,e_{t-1}}) \\
\boldsymbol{h} &= \tanh(W_{mh}\boldsymbol{m} + \boldsymbol{b}_h) \\
\boldsymbol{s} &= W_{hs}\boldsymbol{h} + \boldsymbol{b}_s \\
\boldsymbol{p} &= \mathrm{softmax}(\boldsymbol{s})
\end{aligned} \tag{42}$$

In the first line, we obtain a vector $\boldsymbol{m}$ representing the context e_{i-n+1}^{i-1} (in the particular case above, we are handling a tri-gram model so $n = 3$). Here, M is a matrix with $|V|$ columns, and L_m rows, where each column corresponds to an L_m-length vector representing a single word in the vocabulary. This vector is called a **word embedding** or a **word representation**, which is a vector of real numbers corresponding to particular words in the vocabulary.[15] The interesting thing about expressing words as vectors of real numbers is that each element of the vector could reflect a different aspect of the word. For example, there may be an element in the vector determining whether a particular word under consideration could be a noun, or another element in the vector expressing whether the word is an animal, or another element that expresses whether the word is countable or not.[16] Figure 13 shows an example of how to define parameters that allow you to look up a vector in DyNet.

The vector $\boldsymbol{m}$ then results from the concatenation of the word vectors for all of the words in the context, so $|\boldsymbol{m}| = L_m * (n-1)$. Once we have this $\boldsymbol{m}$, we run the vectors through a hidden layer to obtain vector $\boldsymbol{h}$. By doing so, the model can learn combination features that reflect information regarding multiple words in the context. This allows the model to be expressive enough to represent the more difficult cases in Figure 6. For example, given a context is "cows eat", and some elements of the vector $M_{\cdot,\mathrm{cows}}$ identify the word as a "large farm animal" (e.g. "cow", "horse", "goat"), while some elements of $M_{\cdot,\mathrm{eat}}$ corresponds to "eat" and all of its relatives ("consume", "chew", "ingest"), then we could potentially learn a unit in the hidden layer $\boldsymbol{h}$ that is active when we are in a context that represents "things farm animals eat".

[15]For the purposes of the model in this chapter, these vectors can basically be viewed as one set of tunable parameters in the neural language model, but there has also been a large amount of interest in learning these vectors for use in other tasks, which is outlined in Section 5.6.

[16]In reality, it is rare that single elements in the vector have such an intuitive meaning unless we impose some sort of constraint, such as sparsity constraints [19].

Next, we calculate the score vector for each word: $\boldsymbol{s} \in \mathbb{R}^{|V|}$. This is done by performing an affine transform of the hidden vector $\boldsymbol{h}$ with a weight matrix $W_{hs} \in \mathbb{R}^{|V| \times |\boldsymbol{h}|}$ and adding a bias vector $\boldsymbol{b}_s \in \mathbb{R}^{|V|}$. Finally, we get a probability estimate $\boldsymbol{p}$ by running the calculated scores through a softmax function, like we did in the log-linear language models. For training, if we know e_t we can also calculate the loss function as follows, similarly to the log-linear model:

$$\ell = -log(p_{e_t}). \tag{43}$$

DyNet has a convenience function that, given a score vector $\boldsymbol{s}$, will calculate the negative log likelihood loss:

```
loss = dy.pickneglogsoftmax(s, e_t)
```

The reasons why the neural network formulation is nice becomes apparent when we compare this to n-gram language models in Section 3:

Better generalization of contexts: n-gram language models treat each word as its own discrete entity. By using input embeddings M, it is possible to group together similar words so they behave similarly in the prediction of the next word. In order to do the same thing, n-gram models would have to explicitly learn word classes and using these classes effectively is not a trivial problem [7].

More generalizable combination of words into contexts: In an n-gram language model, we would have to remember parameters for all combinations of $\{\text{cow}, \text{horse}, \text{goat}\} \times \{\text{consume}, \text{chew}, \text{ingest}\}$ to represent the context "things farm animals eat". This would be quadratic in the number of words in the class, and thus learning these parameters is difficult in the face of limited training data. Neural networks handle this problem by learning nodes in the hidden layer that can represent this quadratic combination in a feature-efficient way.

Ability to skip previous words: n-gram models generally fall back sequentially from longer contexts (e.g. "the two previous words e_{t-2}^{t-1}") to shorter contexts (e.g. "the previous words e_{t-1}"), but this doesn't allow them to "skip" a word and only reference for example, "the word two words ago e_{t-2}". Log-linear models and neural networks can handle this skipping naturally.

5.6 Further Readings

In addition to the methods described above, there are a number of extensions to neural-network language models that are worth discussing.

Softmax approximations: One problem with the training of log-linear or neural network language models is that at every training example, they have to calculate the large score vector $\boldsymbol{s}$, then run a softmax over it to get probabilities. As the vocabulary size $|V|$ grows larger, this can become quite time-consuming. As a result, there are a number of ways to reduce training time. One example are methods that sample a subset of the vocabulary $V' \in V$ where $|V'| << V$, then calculate the scores and approximate the loss over this smaller subset. Examples of these include methods that simply try to get the true word e_t to have a higher score (by some margin) than others in the subsampled set [9] and more probabilistically motivated methods, such as **importance sampling** [5] or **noise-contrastive estimation** (NCE; [18]). Interestingly, for other objective functions such as linear regression and special variety of softmax called the **spherical softmax**, it is possible to calculate the objective function in ways that do not scale linearly with the vocabulary size [28].

Other softmax structures: Another interesting trick to improve training speed is to create a softmax that is structured so that its loss functions can be computed efficiently. One way to do so is the class-based softmax [12], which assigns each word e_t to a class c_t, then divides computation into two steps: predicting the probability of class c_t given the context, then predicting the probability of the word e_t given the class and the current context $P(e_t|c_t, e_{t-n+1}^{t-1})P(c_t|e_{t-n+1}^{t-1})$. The advantage of this method is that we only need to calculate scores for the correct class c_t out of $|C|$ classes, then the correct word e_t out of the vocabulary for class c_t, which is size $|V_{c_t}|$. Thus, our computational complexity becomes $O(|C| + |V_{c_t}|)$ instead of $O(|V|)$.[17] The hierarchical softmax [17] takes this a step further by predicting words along a binary-branching tree, which results in a computational complexity of $O(\log_2|V|)$.

Other models to learn word representations: As mentioned in Section 5.5, we learn word embeddings M as a by-product of training our language models. One very nice feature of word representations is that language models can be trained purely on raw text, but the resulting representations can capture semantic or syntactic features of the words, and thus can be used to effectively improve down-stream tasks that don't have a lot of annotated data, such as part-of-speech tagging or parsing [25].[18] Because of their usefulness, there have been an extremely large number of approaches proposed to learn different varieties of word embeddings, from early work based on distributional similarity and dimensionality reduction [23, 26] to more recent models based on predictive models similar to language models [25, 16], with the general current thinking being that predictive models are the more effective and flexible of the two [2].[19]The most well-known methods are the continuous-bag-of-words and skip-gram models implemented in the software `word2vec`[20] , which define simple objectives for predicting words using the immediately surrounding context or vice-versa. `word2vec` uses a sampling-based approach and parallelization to easily scale up to large datasets, which is perhaps the primary reason for its popularity. These methods are not language models, as they do not calculate a probability of the sentence $P(E)$, but many of the parameter estimation techniques can be shared.

[17] *Question: What is the ideal class size to achieve the best computational efficiency?*

[18] Manning (2015) called word embeddings the "Sriracha sauce of NLP", because you can add them to anything to make it better `http://nlp.stanford.edu/~manning/talks/NAACL2015-VSM-Compositional-Deep-Learning.pdf`

[19] Daumé III (2016) called word embeddings the "Sriracha sauce of NLP: it sounds like a good idea, you add too much, and now you're crying" `https://twitter.com/haldaume3/status/706173575477080065`

[20] `https://code.google.com/archive/p/word2vec/`

5.7 Exercise

In the exercise for this chapter, we will use `DyNet` to construct a feed-forward language model and evaluate its performance.

Writing the program will entail:

- Writing a function to read in the data and (turn it into numerical IDs).
- Writing a function to calculate the loss function by looking up word embeddings, then running them through a multi-layer perceptron, then predicting the result.
- Writing code to perform training using this function.
- Writing evaluation code that measures the perplexity on a held-out data set.

Language modeling accuracy should be measured in the same way as previous exercises and compared with the previous models.

Potential improvements to the model include tuning the various parameters of the model. How big should $\boldsymbol{h}$ be? Should we add additional hidden layers? What optimizer with what learning rate should we use? What happens if we implement one of the more efficient versions of the softmax explained in Section 5.6?

References

[1] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] Marco Baroni, Georgiana Dinu, and Germán Kruszewski. Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 238–247, Baltimore, Maryland, June 2014. Association for Computational Linguistics.

[3] Claus Bendtsen and Ole Stauning. Fadbad, a flexible c++ package for automatic differentiation. *Department of Mathematical Modelling, Technical University of Denmark*, 1996.

[4] Yoshua Bengio, Holger Schwenk, Jean-Sébastien Senécal, Fréderic Morin, and Jean-Luc Gauvain. Neural probabilistic language models. In *Innovations in Machine Learning*, volume 194, pages 137–186. 2006.

[5] Yoshua Bengio and Jean-Sébastien Senécal. Adaptive importance sampling to accelerate training of a neural probabilistic language model. *IEEE Transactions on Neural Networks*, 19(4):713–722, 2008.

[6] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A CPU and GPU math compiler in Python. In *Proc. 9th Python in Science Conf*, pages 1–7, 2010.

[7] Peter F. Brown, Peter V. deSouza, Robert L. Mercer, Vincent J. Della Pietra, and Jenifer C. Lai. Class-based n-gram models of natural language. *Comput. Linguist.*, 18(4):467–479, 1992.

[8] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Technical report, Idiap, 2002.

[9] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12:2493–2537, 2011.

[10] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[11] Yoav Goldberg. A primer on neural network models for natural language processing. *arXiv preprint arXiv:1510.00726*, 2015.

[12] Joshua Goodman. Classes for fast maximum entropy training. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 561–564. IEEE, 2001.

[13] Andreas Griewank. Automatic differentiation of algorithms: theory, implementation, and application. In *proceedings of the first SIAM Workshop on Automatic Differentiation*, 1991.

[14] Robin J Hogan. Fast reverse-mode automatic differentiation using expression templates in c++. *ACM Transactions on Mathematical Software (TOMS)*, 40(4):26, 2014.

[15] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[16] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[17] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 27th Annual Conference on Neural Information Processing Systems (NIPS)*, pages 3111–3119, 2013.

[18] Andriy Mnih and Yee Whye Teh. A fast and simple algorithm for training neural probabilistic language models. *arXiv preprint arXiv:1206.6426*, 2012.

[19] Brian Murphy, Partha Talukdar, and Tom Mitchell. Learning effective and interpretable semantic models using non-negative sparse embedding. In *Proceedings of the 24th International Conference on Computational Linguistics (COLING)*, pages 1933–1950, 2012.

[20] Masami Nakamura, Katsuteru Maruyama, Takeshi Kawabata, and Kiyohiro Shikano. Neural network approach to word category prediction for English texts. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING)*, 1990.

[21] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017.

[22] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[23] Hinrich Sch utze. Word space. 5:895–902, 1993.

[24] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015.

[25] Joseph Turian, Lev Ratinov, and Yoshua Bengio. Word representations: a simple and general method for semi-supervised learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 384–394. Association for Computational Linguistics, 2010.

[26] Peter D Turney and Patrick Pantel. From frequency to meaning: Vector space models of semantics. *Journal of Artificial Intelligence Research*, 37:141–188, 2010.

[27] Ashish Vaswani, Yinggong Zhao, Victoria Fossum, and David Chiang. Decoding with large-scale neural language models improves translation. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1387–1392, 2013.

[28] Pascal Vincent, Alexandre de Brébisson, and Xavier Bouthillier. Efficient exact gradient update for training deep networks with very large sparse targets. In *Proceedings of the 29th Annual Conference on Neural Information Processing Systems (NIPS)*, pages 1108–1116, 2015.

[29] R.E. Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.