Chapter – 4

**Neural Networks for Machine Learning**

**Geoffrey Hinton**

with **Nitish Srivastava** & **Kevin Swersky**

**Techniques to get**

**Outputs**

Lectures: Geoffrey Hinton

Learning to predict the next word

A brief diversion into Cognitive Science

The SOFTMAX output function

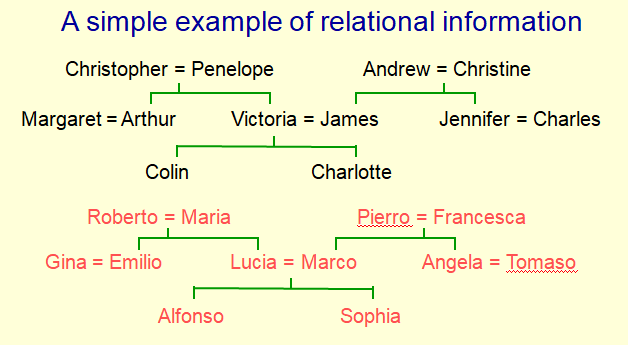
Neuro Probabilistic Language Models

Dealing with many possible outputs

**4.1 Learning to predict the next word**

In this section we'll use the Back-Propagation Algorithm to learn a Feature Representation of the meaning of the word.

* We'll use a simple case, it illustrates the idea about how we can take some relational information, and using the ***back propagation*** ***algorithm*** we turn that information into feature vectors that capture the meanings of words.
* Following diagram shows a simple family tree, in which, for example, ***Christopher*** *and* ***Penelope*** *marry*, and have *children* ***Arthur*** and ***Victoria***.
* We'd like is to train a neural network to *understand* the *information* in this family tree.
* We've also given it (the NN) another family tree of *Italian people* which has pretty much the same structure as the English tree.
* When the NN tries to learn both sets of facts, it's going to be able to take advantage of that analogy.



* **Another way to express the same information**
* The *information* in these *family trees* can be expressed as a ***set of propositions***. If we make up names for the relationships depicted by the trees.
* We'll use the relationships son-daughter, nephew-niece, father-mother, uncle-aunt, brother-sister and husband-wife.
* Make a set of propositions using the 12 relationships:
* son, daughter, nephew, niece, father, mother, uncle, aunt
* brother, sister, husband, wife
* Using those relationships we can write down a set of triples such as:
* (Colleen has-father James)*,*
* (Colleen has-mother Victoria) *and*
* (James has-wife Victoria). This follows from the two above.
* And in the nice simple families depicted in the diagram, the *third proposition* follows from the *previous two*. Similarly, the *third proposition* in the *next set* follows from the *previous two*.
* (charlotte has-brother colin)
* (victoria has-brother arthur)
* (charlotte has-uncle arthur) this follows from the above
* **A Relational Learning task**

So the relational learning task, (i.e. the task of learning the information in those family trees), can be viewed as figuring out the regularities in a large set of triples that express the information in those trees.

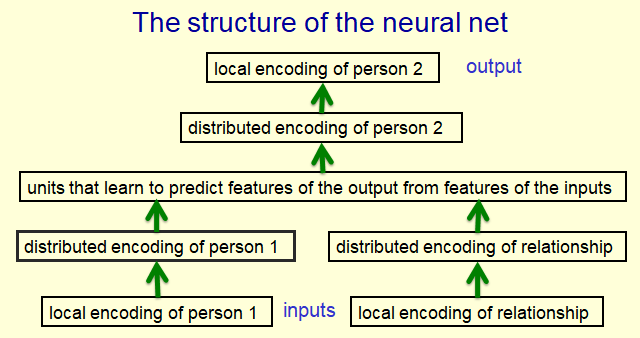
* The obvious way to express regularities is as symbolic rules. For example:

(x has-mother y) & (y has-husband z) => (x has-father z)

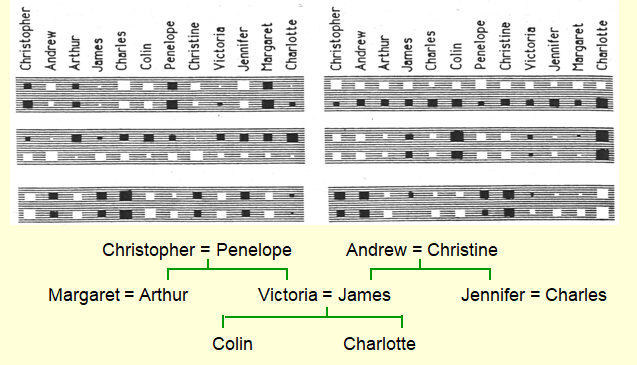
* Finding the symbolic rules involves a difficult search through a very *large discrete space of possibilities*.
* We could search for such rules, but this would involve a search through quite a large space, a ***combinatorially large space, of discrete possibilities***.
* Can a neural network capture the same knowledge by searching through a continuous space of weights?
* **The structure of the Neural Net**

A very different way to try and capture the same information is to use a *neural network* that *searches* through a *continuous space* of ***real valued weights*** to try and capture the information.

* We're going to say it's ***captured the information*** if it can predict the third terminal triple from the first two terms.



* So at the bottom of this diagram here, we're going to input a person and a relationship and the information is going to flow forwards through this neural network.
* The neural network (after it's learned) is output the person who's related to the first person by that relationship.
* The ***architecture*** of this ***net*** was designed by-hand, we decided how many layers it should have. And we also decided where to put bottle-necks to force it to learn interesting representations. So what we do is: We encode the information in a neutral way.
* There are **24** ***possible people***. So the ***block*** at the ***bottom of the diagram*** that says, local encoding of person one, has 24 neurons, and *exactly one of those* will be turned on *for each* ***training******case***.
* Similarly there are **12** ***relationships***. And *exactly one of the* ***relationship*** ***units*** will be turned on.
* And then for a relationship that has a unique answer, we would like to turn on *one of the 24 people at the top-block*, to represent the answer.
* By *using a representation* in which exactly one of the neurons is on, we don't accidentally give the network any similarities between people.
* All pairs of people are equally dissimilar.
* So, we're not cheating by giving actual-information to the network. The people, as far as the network is concerned, are uninterpreted symbols.
* In the next layer of the network, we've taken the local encoding of person 1, and we've ***connected*** it to a ***small set of neurons***, actually ***6 neurons*** for this.
* And because there are ***24 people***, it can't possibly *dedicate one neuron to each person*. It has to ***re-represent*** the ***people*** as patterns of activit***y*** over those six neurons.
* What we're hoping: When it learns these propositions, the way in which it encodes a person, in that distributive panel of activity, will reveal structuring the task or structuring the domain.
* So what we're going to do is: We're going to train it up on 112 of these propositions.
* We'll go through the 112 propositions many times.
* Slowly changing the weights as we go, using back propagation.
* After training, we're gonna look at the six units in that layer that says "distributed encoding of person **1**" to see what they are doing.



* In the layer that says "distributed encoding of person **1**", there are six neurons. And we're looking at the incoming weights of each of those six neurons.
* Here are those six units (in that layer that says "distributed encoding of person **1**") as the ***big gray blocks***.
* We laid out the 24 people, with the ***twelve English people*** in a row along the ***top***, and the ***twelve Italian people*** in a row ***underneath***.
* So each of these blocks has 24 blobs in it.
* The blobs indicate the incoming weights for one of the hidden units in that layer.
* If you look at the Big Gray Rectangle (unit) on the *top right*, you'll see an ***interesting structure*** to the ***weights***.
* The weights along the top that come from English people are all positive. And the weights along the bottom are all negative (indicates Italian).
* That means this unit tells you *whether* the input person is English or Italian. We never gave it that information explicitly.
* Obviously, it's ***useful information***, because in the ***family trees*** that we're learning about, if the input person is English, the output person is always English.
* Knowing that someone is English or Italian already allows you to predict one bit of information about the output. It halves the number of possibilities.

|  |  |
| --- | --- |
| * Now we look at the second unit on the right, has four big *positive weights* at the beginning *top-row & bottom-row*. Those correspond to Christopher and Andrew and their *Italian equivalents*. |  |
| * This unit also has some smaller weights. |  |

|  |  |
| --- | --- |
| * Then it has two big negative weights in top-row and bottom-row, that correspond to Collin, or his Italian equivalent. |  |
| * Then there's ***four*** more ***big positive weights***, corresponding to Penelope or Christina, or their Italian equivalents. * And right at the end, there's two big negative weights, corresponding to Charlotte, or her Italian equivalent. |  |

* By now you've probably realized that, *this specific neuron (unit) represents* what Generation ***somebody*** ***is***.
* It has big positive weights to the oldest generation, big negative weight to the youngest generation, and ***intermediate weights*** which are ***roughly zero*** to the ***intermediate generation***.
* So this unit represents a three-value feature, and it's telling you the Generation ***of the person***.
* Finally, if you look at the bottom gray rectangle on the left hand side, you'll see that has a different structure. If we look at the top row and we look at the negative weights (i.e. black Blobs) in the top row of that unit.
* It has a ***negative weight*** to Andrew, James, Charles, Christine and Jennifer.
* If you look at the English family tree you'll see *Andrew*, *James*, *Charles*, *Christine*, and *Jennifer* are all in the right hand branch of the family tree.
* So, this unit (neuron) has learned to represent which branch of the family tree someone is in.

Again, it's a very useful feature to have for predicting the ***output-person***, because if you know it's a close family relationship, you expect the output to be in the *same branch of the family tree* as the input.

* **What the network learns**
* So the networks in the bottleneck (six hidden units connected to the input representation of person 1) have learned to represent features of people that are useful for predicting the answer.
* We didn't tell it anything about what features (Nationality or Brunch of family tree or Generation) to use.
* It figured out that those are good features for *expressing* the *regularity* in this *domain*.
* Of course, those features are only useful if the *other bottlenecks*, the *one for* relationships, and the *one near the top of the network* before the output person, use similar representations.
* The central layer is able to say *how the* features of the input person and the features of the relationship predict the features of the output person.
* For *example* if the *input person* is a *generation three*, and the *relationship* requires the *output person* to be *one generation up*, then the *output* *person* is a *generation two*.
* But notice, *to capture that rule*, you have to ***extract appropriate features*** at the first hidden layer, and the last hidden layer of the network.
* Also you have to make the ***units in the middle***, relate those features correctly.
* **Another way to see that it works**

Another way to see that the network works, is to *train it on all* but a *few of the triples* and see if it can ***complete*** those ***triples correctly*** (does it generalize?).

* Train the network on all but 4 of the triples that can be made using the 12 relationships. There's **112** **triples**, and we trained it on **108** of them and tested it on the remaining four.
* It needs to *sweep through* the *training set many times* adjusting the weights slightly each time.
* We did that several times and it got either two or three of those right.
* We test it on the 4 held-out cases. It gets about 3/4 correct. That's not so bad for a 24 way choice, it's true it makes mistakes, but it didn't have much training data, there's not *enough triples* in this domain to really *nail down the regularities* very well. It does much better than chance.
* **A large-scale example**

On much *bigger datasets* we can train on a *much smaller fraction* *of the data*. If you train the Network on a much bigger data set, it can generalize from a much smaller fraction of the data. So if you have *thousands and thousands of relationships* you only need to show a small percentage before it can start guessing the other ones correctly.

That research was done in the 1980s, and was a way of showing that Back-Propagation could learn interesting features.

* Suppose we have a database of millions of relational facts of the form (A R B) means: *"A has relationship R to B"*. We could imagine training a NN to discover *feature vector representations of A and R*, that allow it to predict the *feature vector representation of B*.
* If we did that, it would be a very good way of cleaning a database.
* We could train a net to discover feature vector representations of the terms that allow the third term to be predicted from the first two.
* Then we could use the trained net to find very unlikely triples. These are good candidates for errors in the database.
* It wouldn't necessarily be able to make perfect predictions. But it could find things in the database that it thought were highly implausible.
* So if the database contained information, like, for example, Bach was born in 1902. It could probably realize that was wrong, because Bach's a much older kind of person, and everything else he's related to is much older than 1902.
* Instead of actually using ***the first two terms to predict the third term***, we could use the ***whole set of terms***, three of them in this case (all three terms as input), and predict the probability that the fact is correct.
* To train a net to do that, we'd need examples of a *whole bunch of correct facts*, and we'd ask it to give a *high output*.
* To train such a net we need a good source of false facts: We'd also need a good source of incorrect facts, and we'd ask it to give a low output when we're told it was something that was false.

**4.2 A brief diversion into COGNITIVE SCIENCE**

We're now going to talk a little bit about an issue that should interest a cognitive scientist, but may not be of much *interest* to *engineers*.

* In cognitive science, there's been a debate going on for maybe a 100 years about *"the relationship between feature vector representations of concepts"* and *"representations of concepts by their relations to other concepts"*.
* And the learning algorithm we've just seen for Family Trees Example has a lot to say about that debate.
* **What the family trees example tells us about CONCEPTS**

We're now going to make a brief diversion into Cognitive Science. There's been a long debate in cognitive science between two rival theories of ***what it means to have a concept***.

* The FEATURE theory: The feature theory says a ***concept*** *is a big set of* ***semantic features***.
* This is good for explaining ***similarities*** between ***concepts***.
* It's convenient for things like machine learning, because we like to deal with vectors of activities.
* The STRUCTURALIST theory: The structuralist theory says that the ***meaning of a concept*** *lies in its* ***relationships*** *to* ***other concepts***.
* So ***conceptual knowledge*** is best ***expressed*** *not as a* ***big vector***, but as a ***relational graph***.
* In the early 1970s, Marvin Minsky use the *limitations of perceptrons* as *evidence against* feature vectors in favor of relational graph representations.
* **Both sides are wrong**
* But both sides *in this debate are* wrong because *both sides believe that the two theories are rivals and* ***they're not rivals at all***.
* A Neural Net can use vectors of semantic features to implement a relational graph.
* For example, in the neural network that learns family trees, we can think of explicit inference as: "I give you person\_1 and I give you a relationship then you tell me person\_2.
* No *explicit inference* is required to arrive at the intuitively obvious consequences of the facts that have been *explicitly* *learned*.
* And to arrive at that *conclusion*, the neural net doesn't follow a whole bunch of *rules of inference* (It can "intuit" the answer in a forward pass). It just *passes information forward through the net*.
* As far as the neural net is concerned, the answer is intuitively obvious.
* If you look at the details of what's happening, there's lots of probabilistic features that are influencing each other. We call these microfeatures to sort of emphasize *they're not like* ***explicit conscious features***.
* In a ***real brain***, there might be *millions of interactions*, and as a result of all these interactions, we can make one step of explicit inference.
* And that's what we believe is involved in just seeing the answer to something. There are *no intervening conscious steps*, but nevertheless there's a *lot of computation* going on in the *interactions of neurons*.
* So we may use explicit rules for *conscious deliberate reasoning*, but a lot of our *common sense reasoning*, particularly *analogical* *reasoning*, works by just seeing the answer, with no conscious intervening steps.
* Even when we are using *explicit rules* to do *conscious reasoning*, we need to just see *which rules to apply*, in order to avoid an infinite regress.
* **Localist and distributed representations of concepts**

The obvious way to implement a ***relational graph*** in a *neural net* is to treat a neuron as a node in the graph and a connection as a binary relationship. But this "localist" method will not work.

* So, many people, when they think about ***implementing a relational graph*** in a Neural Net, just assume that you should make:

1. A neuron correspond to a node in the relational graph and
2. A connection between, two neurons correspond to a binary relationship.

* But this method doesn't work, we need many ***different types*** of ***relationship*** and the ***connections*** in a neural net do not have discrete labels.
* For a start, the relationships come in different flavors. They're different kinds of relationship like: mother of, or aunt of.
* Also a connection in a neural net only has a strength. It doesn't come in different types.
* Also we need to deal with ternary relationships as well as binary ones like: 'a' is between' b' and'c'.
* We still don't know the right way to implement *relational knowledge* in a neural net.
* Many to many relations: But many neurons are probably used for each concept and each neuron is probably involved in many concepts. This is called a "Distributed Representation".
* But it seems very probable that many *neurons* are used for *representing* *each of the concepts* we know, and
* Each of those neurons is probably involved in dealing with many different concepts.

This is called a Distributed Representation. It's a many to many mapping between *concepts* and *neurons*.

**4.3 The SOFTMAX output function**

In this section we'll discuss about the Soft-Max output function.

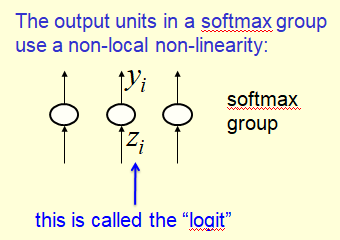
* Soft-max: This is a way of ***forcing*** the ***outputs*** of a neural network to ***sum to 1*** so they can represent a probability distribution across ***discrete mutually exclusive (disjoint)*** alternatives.
* **Problems with Squared Error**

Before we get back to the issue of how we learn feature vectors to represent words, we're gonna have one more digression, this time it's a *technical diversion*.

* So far we've talked about using a Squared-Error-Measure for training a Neural Net and for linear neurons it's a *sensible thing to do*. But the squared error measure has some drawbacks.
* For example: If the desired output (target) is **1** and the actual output of a neuron is **0.00000001**, then there is almost no gradient for a logistic unit to fix up the error.
* It's way out on a plateau where the slope is *almost exactly horizontal*. So, it will take a very, *very long time* to change its ***weights***, even though it's making almost as big ***an error*** as it's possible to make.
* If we are trying to assign probabilities to ***mutually exclusive class labels***, we know that the outputs should sum to 1, but we are *depriving* the network of this knowledge.
* Any answer in which we say, "the probability of A is **3/4** and the probability of B is also **3/4**, is just a ***crazy answer***".
* And we ought to tell the network that information: we shouldn't *deprive* it of the knowledge that these are mutually exclusive answers.
* So the question is: Is there a different cost function that will work better? Is there a ***way of telling*** it that these are ***mutually exclusive*** and then using a, an ***appropriate cost function***?
* The answer is: There is. What we need to do is: *force the* ***outputs*** *of the* ***neural net*** to represent a ***probability distribution*** across ***discrete alternatives***.
* **SoftMax**

The way we do this is by using something called a soft-max. It's a kind of ***soft continuous version*** of the Maximum Function.

* The way the units in a soft-max group work is that: they each *receive* some ***total input*** they've accumulated from the ***layer below*** i.e. for the i-th unit, and that's called the ***logit***.
* Then they give an output that *doesn't just depend* on their own , it depends on the Zs - ***accumulated*** by their ***rivals*** as well.



* So we say that the output of the i-th neuron is divided by the ***sum of that same quantity*** for all the *different neurons* in the *softmax-group*.
* And because the *bottom line* of that *equation* is the *sum of the top line* over ***all possibilities***.
* We know that when you add over all possibilities you'll get one i.e, the sum of all the 's must come to one, .
* And each *have to lie between* zero and one .

So we force the s to *represent* a ***probability distribution*** over ***mutually exclusive alternatives*** just by using that soft-max equation.

|  |  |
| --- | --- |
| * The soft max equation has a nice simple derivative. If you ask about how the changes as you change the , that obviously depends on, all the other s. * Also the itself depends on all the other s. And you get a nice simple form, just like you do for the logistic unit, where the derivative of the ***output w.r.to the*** ***input***, for an individual neuron in a softmax group, is just: * It's not totally trivial to derive that. If you tried differentiating the equation , you must remember that things turn up in that normalization term on the bottom row. It's very easy to forget those terms and get the wrong answer. |  |

* **CROSS-ENTROPY: the right cost function to use with softmax**

Now the question is, if we're using a softmax group for the outputs, what's the *right cost function*?

* As an answer, is that the *most appropriate cost function* is the ***negative log probability of the correct answer***. i.e., we want to maximize the log probability of getting the answer right.
* i.e. the right cost function is the negative log probability of the right answer.
* So if *one of the target values* is a **1** and the *remaining ones* are **0**, then we *simply sum of all possible answers*.
* We put **0**s in front of all the wrong answers.
* We put **1** in front of the right answer

|  |  |
| --- | --- |
| * That gets us the negative log probability of the correct answer, as you can see in the equation. * is the "target value" * This equation is called the Cross Entropy Cost Function. * **C** has a very big gradient when the target value is **1** and the output is almost **0**. |  |

* We can see that by considering a couple of cases:
* A value of **0.000001** is ***much better*** than **0.000000001**: A value of one in a million (0.000001) is much better than a value of one in a billion (0.000000001)if the correct answer is **1**.
* The ***steepness*** of **dC/dy** exactly ***balances the flatness*** of **dy/dz**: When you make the output value increased by less than one millionth, the **value of C** improves by a lot. That means it's a very, very steep gradient for **C**.

So value of one in a million (0.000001) is much better than a value of one in a billion (0.000000001), even though it *differs by less* *than a millionth*.

* The **cost function C** has a very steep derivative when the answer is very wrong and that exactly bounces the fact that "the way which the output changes if you change the input, **dy/dz** is very flat when the *answer is very wrong*.

|  |  |
| --- | --- |
| * derivative of CROSS ENTROPY w.r.to the logit: When you multiply the two together: you get the derivative of CROSS ENTROPY w.r.to the logit going into **output unit i**. |  |

* We use the chain rule so that derivative is: ***How fast the cost function changes as you change the output of the unit*** times ***How fast the output of the unit changes as you change*** .
* Notice we need to add up across all the **j**'s, because ***when you change the* i*, the output of all the different units changes***. The result is just the ***actual output*** minus the ***target output*** .
* When the actual target outputs are very different, it has a slope of **+1** or **-1** and the ***slope*** is ***never bigger than*** **+1** or **-1**.
* But the slope never gets small until the two things ( and ) are pretty much the same. In other words, you're getting pretty much the right answer.

**4.4 Neuro Probabilistic Language Models**

Now we're going to look at a practical use for feature vectors that represent words. The uses in Speech Recognition Systems, where having a good idea of *what somebody might say next (prediction)* is very helpful in *recognizing the sounds* they make.

* **A basic problem in speech recognition**

We're now going to see an important practical use for feature vectors that describe words.

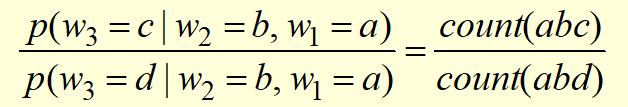
* ***We cannot identify phonemes perfectly in noisy speech:*** When we're trying to do speech recognition, it's impossible to identify phonemes perfectly in noisy speech.
* ***The acoustic input is often ambiguous:*** The *acoustic input just isn't good enough*. It's often ambiguous. There may be *several different words* that *fit* the *acoustic signal* equally well.
* People use their understanding of the meaning of the utterance to hear the right words. We don't notice this a lot of the time, because we're so good at using the meaning of the utterance to hear the right words.
* For example if we speak: "We do this unconsciously when we wreck a nice beach" and you would hear, "We do this unconsciously when we recognize speech".
* You can actually hear the slight difference between ***"wreck a nice beach"*** and ***"recognize speech"***, when there is ***no noise*** in your environment.
* But if you're expecting "recognize speech", particularly if there's ***noise around you*** wouldn't hear "wreck a nice beach".

So we're very good at doing this. We do it all the time when we do it unconsciously.

* That means that ***"speech recognizers"*** have to know ***which words are likely to come next and which are not***. (it's some kind of prediction).
* Fortunately, words can be predicted quite well without having a full understanding of what's being said.
* **The standard "trigram" method**

So there's a standard method for predicting the ***probabilities of the various words*** that might come next, it's called the trigram method.

* You take a ***huge amount of text*** and you ***count the frequencies*** of all triples of words. Then you use these frequencies to make bets on the relative probabilities of the ***next word*** *given the* ***previous two*** ***words***.
* So if we've heard the words ***a*** and ***b***. We can look at the ***counts*** that we have in our ***huge body of text***. For the sequence **abc**, and the sequence **abd**, we can say that the relative probability that the third word will be c versus the third word will be d, is given by the ratio of the two counts: **abc** and **abd**.

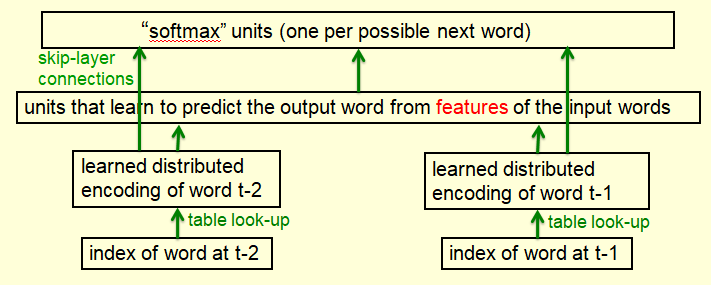


* ***But this method has very limited use:*** Until very recently, this was the state of the art method for ***getting the probability*** of the ***next*** ***word*** to help out the ***speech recognizer***.
* We can't use *much* ***bigger contexts*** *than* ***two previous words***, because there are just ***too many possibilities to store***, and if we did use bigger contexts, the counts would be mostly zero.
* Even for ***two word contexts***, there's many contexts that you will never have heard. For example, if we say "dinosaur pizza", that's probably a ***string of two words*** *that you've* ***never heard*** *before*.
* We have to "back-off" to digrams when the count for a trigram is too small: For cases like that, we have to **"back-off"** to ***individual words***. So, after "dinosaur pizza", you predict the next word by *just seeing* what's likely to come after the word "pizza", because you've never heard "dinosaur pizza" before.
* The ***probability is not zero*** just because the ***count is zero*** : What you mustn't do is to say that "the probability's a zero" just because you haven't seen an example. That's clearly not true.
* **Information that the trigram model fails to use**

The trigram model *fails* to *use a lot of obvious information* that will help you *predict* the *next word*.

* Suppose for example, you have seen the sentence "the cat got squashed in the garden on friday". That should help you predict the words in the sentence, "the dog got flattened in the yard on Monday".
* In particular, the trigram model doesn't understand the similarities between words like: cat/dog, or squashed/flattened, or garden/yard, or Friday/Monday.
* So it can't use it's past experience with one of those words to help it with the other one.
* To overcome this limitation, what we need to do is ***convert the words*** into a ***vector of*** semantic ***and*** syntactic ***features***. And use the ***features of previous words*** to predict the ***features of the next word***.
* Using a feature representation also allows us to use *much bigger context*, that contains many more words. (Eg: 10 previous words).
* **Bengio’s Neural Net for predicting the next word**

Yoshua Bengio's approach for *language models*, and his *initial network* for doing this, is actually very similar to the ***family trees network***. It's just applied to a real problem, and is much bigger.



* At the ***bottom*** you can think of us as *putting* in the ***index of a word***, and you could think of that as a set of neurons of which *just one is on*.
* The weight from that on neuron will *determine the* ***pattern*** *of* ***activity*** in the *next* ***hidden*** *layer*.
* So the weights from the active neurons in the bottom layer will give you the ***pattern of activity*** in the layer that has the ***distributed*** ***representation of the word*** (i.e. it's feature vector).
* But this is just equivalent to saying: "you do table look-up". You have a ***stored feature vector*** for ***each word***, and with *learning*, you modify that feature vector.
* Which is exactly equivalent to modifying the weights coming from a single active-input unit.
* After getting ***distributed representations*** of a few ***previous words***, (I've only shown two here but you would typically use, say, five).
* You can then, use those *distributed representations*, via ***hidden layer***, to predict, via huge softmax, what the ***probabilities*** are for all the various words that might come ***next***.
* Use skip-layer connections that go straight from the *input words* to the *output words* make extra refinement that makes it work better.
* Because the ***individual input words*** are, are individually quite ***informative*** about what the ***output word*** might be.

***Bengio's model*** was *slightly worse* than ***Trigram's*** *that predicting next word*, but if you ***combined*** it with ***Trigram's*** it improved things.

Since then these language models that use ***feature vectors*** for words have been improved considerably, and they're now considerably better than trigram models.

* **A problem with having 100,000 output words**

One problem with having a **big SoftMax output layer**, is that you might have to deal with *100,000 different* ***output words***. Because typically in these language models,

* The plural of a word is a *different word* from the ***singular***.
* The various different tenses of a verb are *different words* from ***other tenses***.
* So each unit in the last hidden layer of the Neural Net, might have to have a ***Hundred-Thousand*** outgoing weights.
* ***We cannot afford to have many hidden units:*** We can only afford to have a few units there before we start over-fitting. Unless we have a huge number of training cases. Organization like Google might have so much training data that it can afford to have a very big softmax layer.
* ***Last hidden layer small:*** We could make the *last hidden layer small*, but then it's *hard to get the 100,000 probabilities* ***right***.
* We could try and make the last hidden layer small, so we don't need too many weights. But then we have the problem that *"we have to get the* ***100,000 probabilities*** *of the various words that might come next,* ***fairly accurately right****"*.
* ***The small probabilities are often relevant:*** It's not just the *big probabilities* we need. A *very large number of words* will have *small probabilities*.
* The small probabilities are often relevant. It might be that the speech recognizer has to decide *"whether it's two different* ***rare*** *words"*, and then it's very relevant *which of those* is *more common* in the *context*, even though both of them are pretty *unlikely*.

The question is, ***Is there a better way to deal with such a large number of outputs?*** We'll see several ways in the next section.

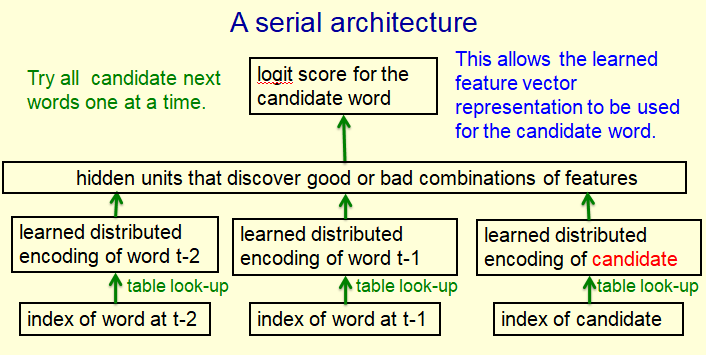
**4.5 Dealing with many possible outputs**

In this section we're going look at *various ways to avoid* having to use *100,000 different output* units in the softmax, if we want to get probabilities for a 100,000 different words.

* At the end of the section, we'll see an example of the words, or the word representations that are learned by a particular method.
* To see what these representations look like, what we do is we **embed** them in a **2D-space**, then, we can see which words have *extremely similar representations*.
* That gives us a ***feel*** for what the ***neural network*** has been ***able to learn***, just in trying to predict the next word or perhaps, the middle word of a *string of words*.
* One way to avoid having a **100,000** output units, is to go *through* the *words* ***one at a time***. So, the ***input*** consists of a ***context*** of ***previous words***, and we plug in a ***candidate word***.
* Then the output of the network is a ***score*** for how good that ***combination is***. How well does that ***candidate word*** fit into that context?
* Of course, that means we have to ***run*** this ***NN*** ***many, many times*** but most of the work can be ***shared*** so the ***inputs*** that come into that ***final hidden layer*** from the context, stay the same for all different candidate words.
* So we only need to run a small part of the network for *each candidate word*. We try all the candidate words one at a time. And we don't want to.
* **A serial architecture**

One way to avoid having a 100,000 different output units, is to use a serial architecture.

* We put-in the context words as before, but now we also put-in a candidate for the next word in the same way as the *context words*.
* Then, we go forward through the net and what we output, is a score for *how good that candidate word is, in that context*.
* Of course, we have to *run forward* through this net, many, many times. But *most of the work* only needs to be done *once*.
* So, the inputs from the context to that *big hidden layer* are the ***same*** for every *different candidate word*.
* The only bit we need to ***run*** for each *candidate word* is: the ***inputs*** coming from the ***candidate word*** and the ***final output*** to the ***score***. And, of course, that doesn't have many weights in it.



* So, we try all the candidate words (next words) one at a time.
* By putting in the word as a candidate at the bottom, we're able to use the learned feature vector for that *candidate word*, that we *learned* when it was a *context word*.
* So, we can have the same *representation* of the word when it's part of the *context*, and when it's a candidate for the *next word* that we're trying to predict.
* **Learning in the serial architecture**

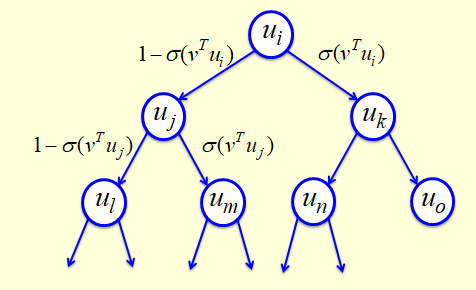
Learning in the serial architecture works in the following way:

* We first *compute the score* (logit score) for each possible candidate word and then we *put all those scores (logits)*, which we *computed sequentially* into a big softmax to get word probabilities.
* Now, the difference between the word probabilities and their target probabilities (which is normally **1** for the correct word and **0** for everything else), gives us the Cross Entropy Error Derivatives and
* We use those *derivatives* to *change the* weights, in such a way that we **raise** the ***score*** for the ***correct candidate*** and we **lower** the ***scores*** for all of it's ***high scoring rivals***.
* We can save a lot of time in this architecture if *instead of considering* ***all possible candidate words***, we only consider ***a small set of candidate words*** *suggested* by some other *predictor*.
* For example, we could use the ***Neural Net*** to revise the ***probabilities* of the words** that the Trigram model thinks are likely.
* **Learning to predict the next word by predicting a path through a tree (Minih and Hinton, 2009)**

A different way to avoid a great big softmax is to *structure the words* into a tree.

* We arrange all of the words in a binary tree, with the words as its leaves,
* Then use the *context of previous words* to generate a "prediction vector" **v**.
* We compare that prediction vector with a vector **u** that we ***learned*** (learned vector **u**) for each ***node*** of the ***tree***.
* The way we compare **v** with a learned vector, **u**, at each node of the tree is: by taking a scalar-product of the prediction vector v and the learned vector u then we apply the logistic function.
* We apply the logistic function to that scalar-product of **u** and **v**, and that will give us the probability of taking the right branch in the tree and *one minus that*, gives us the probability of taking the left branch in the tree.

That's how we predict the probabilities of taking the two branches of the tree.



* So, the tree looks like this and if that sigma is the logistic function, you can see at the top of tree that we take the logistic of the (predictionVectorlearnedVector) at the ***top node***, i.e. is the probability taking the ***right*** *branch*.
* Conversely we take the left branch with minus logistic of the , , i.e. is the probability taking the ***left*** *branch*

And so on, all the way down the tree to the word we want.

|  |  |
| --- | --- |
| * So, when we're learning, we use our contacts to get a prediction vector, we use quite a simple Neural Network, for this work, where we take the ***feature vector*** *for each word* and those feature vectors directly *contribute* evidence in favor of a ***prediction vector***. * We simply ***add*** up the ***evidence*** contributed by those feature vectors and that *gives* us the *prediction vector*. * That *prediction vector (v)* then gets compared with *the vectors that have been learned (u)* for all the nodes in the tree ***on the path*** *to the* ***correct******next******word***. So, that would be nodes **i, j**, and **m** in this tree. * That red path shows you the *path to the word* that actually ***occurred next*** and those are the only nodes we need to consider during learning. |  |

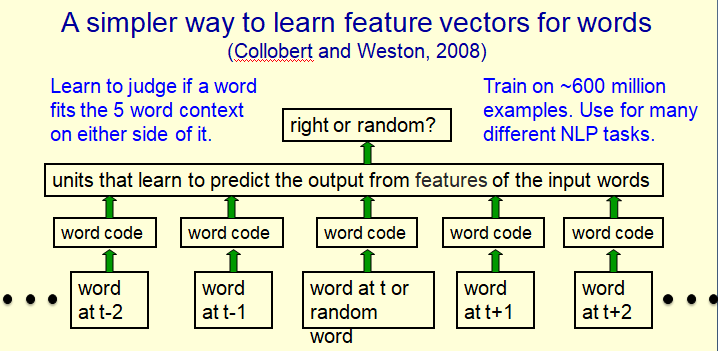
* What we know is that, we'd like the probability of predicting that word to be ***as high as possible***.
* i.e. we'd like the ***probability*** of ***taking that path*** to be as high as possible. Hence, we'd like the ***product of the probabilities*** on the *individual elements* of that path to be as high as possible.
* And finally that means we'd like the sum of their log probabilities to be high.
* **A convenient decomposition**

We can benefit from a nice decomposition here. That when we try and *maximize the* ***probability*** of picking the *correct* ***target word***, we're really trying to maximize the sum of the log probabilities of taking *all the branches* on the path that leads to that *target word*.

* ***Maximizing*** the ***log probability*** of *picking the* ***target word*** is equivalent to ***maximizing*** the ***sum of the log probabilities*** of *taking all the branches* on the *path* that leads to the ***target word***.
* So, *during learning*, we only need to consider the nodes on that correct path. And that's a huge win, that's ***exponentially fewer nodes*** when considering all of the nodes. It's instead of .
* For *each* of those *nodes*, we know the *correct branch*, because we know what the next word is, we know the *current probability* of taking *that branch*, by *comparing* the *prediction vector* with the *learned vector* of the **node**.
* As so, we can get derivatives for learning both the prediction vector **v** and learned vector of that node, **u**.
* This makes *training hundreds of times faster*. Unfortunately, it's still slow at test time. At ***test time***, you *need to know* the *probabilities of many words* to help speech recognizer. And so, you can't just consider ***one path***.
* **A simpler way to learn Feature Vectors for words (Collobert and Weston, 2008)**

There's a much simpler way to learn feature vectors for words. This is done by **Collobert** and **Weston**, what they did was:

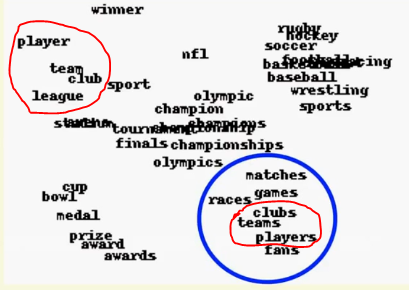
* Learn feature vectors for words and then showed that the feature vectors they learned were very good for a whole bunch of different NLP tasks (natural language processing).
* They're *not trying to predict* the *next word*, they're just *trying to get* ***good feature vectors*** *for* ***words***, and so, they use both the past context and the future context.
* So, they look at a window of eleven words, five in the past and five in the future. And in the middle of that window, they put either the correct word, the one *that actually occurred in the text*, or a random word.
* Then they trained a *Neural Net* to produce an *output* that's ***high*** if it's a ***correct word*** and ***low*** if it's a ***random word***.



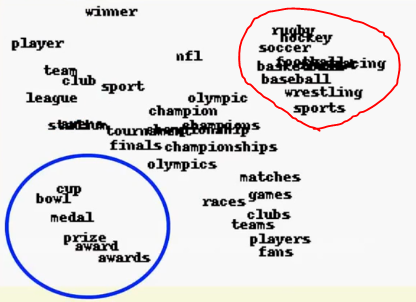
* The Neural Net works the same way as before. We map the *individual words* to feature vectors, these word codes, and then we use the *feature vectors* in the *neural net* (possibly with more layers in the neural net) to try and predict whether this is the right or wrong word for that context.
* So, what they're really doing is judging whether the *middle word* is an *appropriate word* for the *five-word context* on either side of it.
* They trained this up on about *600 million examples* from *Wikipedia*. And they showed that the vectors they get are good for a variety of different natural language processing tasks.
* **Displaying the learned Feature Vectors in a 2-D map**

We can get an idea of the quality of the learned feature vectors by displaying them in a 2-D map.

* We lay-out the word vectors in such a way that very similar vectors are very close to one another. That's how we can see: what words the ***neural network*** thinks have similar meanings.
* We're going to use a ***multi-scale method*** called **"t-sne"**. t-sne is able to putting very similar words close to each other, it's also able to put similar clusters close to each other. So, it gives you structure at many different scales.
* What we see, is that the learned feature vectors capture lots of *subtle semantic distinctions*. And they do this just by looking at strings of words from *Wikipedia*. Nobody tells them anything other than the fact that *those eleven words occurred* in the string.
* There's no extra *supervision*.
* What's remarkable is that, *that contextual information*, the fact that *these words occurred together*, tells you an awful lot about what the word means.
* In fact, some people think that's the main way we learn the meaning of words.
* Here's an example. If I give you a sentence with a word you've never heard before like, ***"She scrammed him with a frying pan"***. From that one sentence, you already have a pretty good idea what "scrammed" means.
* It's conceivable that *she was trying to impress him with her cooking skills* and so scrammed means impressed or something like that (but probably it means something like walloped).
* So, here's *part of a 2-D map* in which we laid out the *2500 commonest words* and you'll see this part of the map is *all about* games. Not only that, it's got ***similar*** kinds of ***words together***.
* Notice, matches, games and races are together. It's got players, teams and clubs, league also together.



* It's got the things you win at games together, like cups, bowls, medals and prizes. It also got different kinds of games together rugby, soccer, football, hockey etc.



* It's done a very good job of taking these words to do with *games* and finding out which ones are very *similar in meaning*. It's using *very* *similar feature vectors* for those words which has similar context. If one word was a good word for a context, the other word's probably also a pretty good word for that context.

|  |  |
| --- | --- |
| * Here's another part of the map. This part of the map is entirely about places. * At the top, it has ***US states***. Under that it has some ***cities*** in ***North America***. And under that it has some ***countries***. * Notice underneath Cambridge, there's something else that's very similar to Cambridge. Also it puts Toronto with Detroit and Ontario and Boston. * ***Toronto's*** in *English-speaking* ***Canada***. And it's put ***Quebec***, which is in *French-speaking* ***Canada***, with ***Berlin*** and ***Paris***. If we look at the bottom, we can see that it thinks ***Iraq*** is pretty similar to ***Vietnam***. |  |

|  |  |
| --- | --- |
| * Here's another example. Notice, these are adverbs. And it's understood that likely and probably and possibly and perhaps, all *mean very similar things*. * It's also understood that entirely, completely, fully, and greatly have very *similar meanings*. * And it's understood various other kinds of similarity. For example, ***which*** and ***that***, or ***whom*** and ***what***, or ***how*** and ***whether*** and ***why***. |  |