Language translators

This is a brief introduction to the techniques of automated language translation which is now becoming a reality in handheld devices. If you have ever tried to learn a foreign language, you will have had to go through the process of learning vocabulary - for example that "cheese"' in French is '"fromage" Then we have to learn the rudiments of grammar, which has been de-emphasised in teaching English in our schools to the point where foreign language teachers have to introduce their students to grammar, and some of our politicians are happy enough to say "That is news to him and I" without a second thought as to whether it is grammatically correct. Then finally we have to understand meaning, common usage, and colloquialisms well enough to carry on a reasonable conversation.

So, how have we arrived at the situation where a tiny handheld device can match a skilled translator and easily outperform a novice? Instead of our human paths to translation, these devices use machine learning with neural networks, which have been trained on very large databases of the same sentences in different languages, and statistical modelling, rather than learning the vocabulary, grammar and semantics that we poor humans must.

Where do such databases exist? For example, the digital corpus of the European Parliament contains the majority of the documents of the Parliament as full documents in the different languages and as sentence aligned versions for language pairs such as French and English.An example of a sentence aligned pair might be " Do you know CPR? aligned with "connalissez-vous le massase cardiaque?" These documents are the feedstock of machine language translators. They contain literally many millions of sentences which are used in the training. Most of humanity would have neither the interest nor the capacity to wade through this amount of material.

The underlying idea of artificial intelligence is the neural network. The mathematical description of a single node (neuron) i of a network takes inputs $x_{i,j}$ from other neurons j, applies a function $f()$, typically involving weights $w_{i,j}$, and a bias b_i, so that

$$output_i = f(\sum_j w_{i,j} x_{i,j} + b_i)$$

. One popular function is the $relu$ function defined as follows:

$$relu(x) = 0; x < 0.0$$
$$relu(x) = x; x > 0.0$$

This neuron "fires" with a strength proportional to its positive input and fails to fire if its input is negative. A major reason for its popularity is its ease of differentiation.

The nodes are usually arranged in layers, with an input layer, consisting of a tokenised sentence in the case of language translation, and a final out output, again in this case consisting of a tokenised sentence in the translated language. Between the input layer and the output layer are a set of "hidden layers'. Each layer will consist of multiple neurons, each with its own set of weights and bias.

Such neural nets were developed last century. Interest waned when it was realised that the computational power to implement efficient nets simply did not exist at the time. The current resurgence of AI in general is due to the spectacular improvement in computational power over the decades and in particular the gpu. The basic idea of "training" a neural net is to take a set of (input, output) pairs - in the case of language translation, input being a sentence in one language and output being its translation in the other, and adjust all the individual weights and biasses so as to minimise the aggregate differences between the output predicted by the neural net and the actual translation of the input.

Various measures of the aggregate discrepancy have been used when machine learning has been applied in different arenas, most of them fairly sophisticated and not easy to encapsulate. A common measure used in machine translation is cross-entropy which you can google for a tutorial explanation. One measure of entropy itself is the number of bits required to transmit information about a variable selected from a probability distribution.

We use this idea of a probability distribution in lots of different contexts.For example, if we were compressing a text file using Huffman coding, we would compute the probabilities of each letter occurring, and then develop a bit code so that the most frequently occurring letters had the shortest bitcodes, and the least frequentlly occurring had the longest bitcodes. Thiis is in contrast to ASCII where the bitcode of each character has the same length, and enables us to compress ASCII text.

Cross entropy calculates the average number of bits needed to transmit data from one distribution, in this case tokens of one language, into another distribution, tokens of the second laguage.

$$H(P,Q) = \sum_{x \in X} P(x) log_2(Q(x))$$

where $P(x)$ is the probability of occurrence of x in distribution P, and $Q(x)$ is the probability of x occurring in the distribution Q.

In the case of binary machine learning classification, we can assign a probabliity distribution $P(x)$ of 1 for the target target identification and 0 for the the others, and obtain a probability distribution $Q(x)$ for possible predicted outputs from the neural network output. For more than two categories we need to apply categorical cross entropy which is yet more complex, but adequate material is available on the net, and machine learning environments such as keras have implementations available to choose.

Part of the machine translation process is to turn sentences into words, and eventually into numbers that can be used in the algebraic training training process . If we start with with the sentence "What's gone wrong for NSW today? " the tokenising process will probably reduce that to something like "<start> what s gone wrong for nsw today ? <end> ". At the start of the training process, the program will have to digest sufficient of these sentences to create a working vocabulary, and then form a dictionary where each word is mapped to an integer, so that the sentence might now map to a sequence something like {1 27 6 15 36 18 2}

Obviously the vocabulary and dictionary size will expand with the number of sentences digested. In training with limited computing resources, words which occur with low frequency might be mapped to a single token <unknown > with just one dictionary entry. This will reduce the training computation time at the expense of the fidelity of translation.

I note in passing that with speech recognition applications such as Apple Siri, there are other steps where the analog sound input has to be processed to produce a word sentence, probably with another neural net, to produce a digital version of the input, and a digital to analog conversion of the digital output to produce a synthetic speech output at the other end of the process.

So, at the end of the sentence ingestion phase, we have a numeric representation of each sentence. If we are doing English to French translation, we have an English vocabulary dictionary, a French vocabulary dictionary, and a huge collection of pairs of sentences with the same meaning - that collection having been produced "manually" by expert translators over the years. The task now is to "train" a neural network using that collection, by adjusting the weights and biasses of each neuron so as to minimise an aggregate loss function, such as the categorical cross entropy mentioned earlier.

Mathematically, what we have is a function of millions of variables to minimise, which is why machine learning has only come into its own since the capability of our computer hardware has reached its present strength. The collection of sentences is split into a larger training set and a test set, still of substantial size. As far as the training process is concerned, the members of the test set are unknown and are used to evaluate the accuracy of the network weights produced by the training process.

Many AI applications share this training step, which is implemented in environments such as google's tensorflow, on hardware such as nvidia's gpu and the steadily expanding range of custom chips aimed at inference processing. The computations associated with each neuron are the simple function evaluations mentioned earlier:

$$output_i = relu(\sum_j w_{i,j} x_{i,j} + b_i)$$

where

$$relu(x) = 0; x < 0.0$$
$$relu(x) = x; x > 0.0$$

which is relatively cheap to compute. This means that once trained, a neural network can process its input quickly, to translate a sentence, or classify an image, or whatever. The training process is what takes the time and computational power, because we will have a large number of (input, output) training examples, often a huge number of weights and biasses, depending on the number and width of the hidden layers, and an aggregate loss function which is the sum over all the training collection of the loss for each training pair, which in turn is a function of all the weights and biasses.

For a function $F(\underline{x})$ the uphill direction at a point $\underline{x}$ in multi-dimensional space is the gradient vector

$$\nabla F = (\frac{\partial F}{\partial x_i})$$

We can lower F by taking a step in the negative gradient direction, with the step magnitude reduced by a factor, called the "learning rate" in machine learning", which typically might be as small as 0.001 per step. Remember that at this stage we have an loss function aggregated over thousands of samples , with hundreds of thousands of weights and biasses, so taking a small step in the negative gradient direction is computationally intensive. Addressing this problem has been the subject of ongoing research, resulting in recommended algorithms such as Adam, which are implemented in machine learning environments such as keras and tensorflow. Firstly the gradient of the aggregate loss function requires the sum of the gradients for the individual members of the training set. Rather than use the aggregate, A simple stochastic algorithm could choose to cycle through the gradients one at a time in a random order and apply the gradient step to the weights and biasses at each step. A refinement of this approach is to compute the aggregate gradient for randomly chosen batches of training examples, to reduce the amount of work involved in each application of the gradient step.A further refinement was the introduction of 'momentum', where the gradient used is a linear combination of gradient currently computed and the one used at the previous step - based on the analogy of keeping a ball rolling downhill. The Adam variant computes individual adaptive learning rates from estimates of the first and second moments of the gradients, and incorporates variable learning rates.

An essential detail of the process is how the the gradients are computed for the individual weights and biasses, in each layer of the network, - backpropagation. If the output of an individual neuron in a layer is

$$a_i = f(\sum w_{i,j} x_j + b_i)$$

where the x_j are the outputs from the previous layer, then for the $relu()$ function

$\frac{\partial a_i}{\partial w_{i,j}} = 0$ for $\sum w_{i,j} x_j + b_i \leq 0$, else x_j.

and

$\frac{\partial a_i}{\partial b_i} = 0$ for $\sum w_{i,j} x_j + b_i \leq 0$, else 1.

. Using the chain rule, backpropragation works its way back from the output layer, computing elements of the gradient function for all the weights and biasses.

Language translation as implemented in tensorflow uses the additional notion of *attention* in the sequence to sequence $seq2seq'$ model. Each of the input words is assigned a weight by the attention mechanism to predict the next word in the sentence, which is then chosen from a *softmax* of possible choices. This is described more fully in the online literature, often using Bahdanau attention model for the encoder.

The input is put through an encoder model which gives an encoder output of shape($batch_size$, max_length, $hidden_size$) with an encoder hidden state of shape ($batch_size$, $hidden_size$). The aim is to predict the next word, based on a context of max_length previous words in the sentence by assigning weights to the previous words in the context.. The whole process is too detailed to explain here, and is well documented in the tensorflow tutorial

`https://www.tensorflow/tutorials/text/nmt_with_attention`

To give some idea of the training effort required, on the nvidia jetson, training to translate between English and French on a subset of $80,000$ paired sentences, mostly simple, using 15 epochs (passes through the entire training set) using batch sizes of 64 pairs, takes an average 8 minutes per epoch, for a total of 2 hours compute time, to reduce the aggregate categorical cross entropy loss function to 0.047 from tts value at the end of the first epoch of 1.5.

The fidelity of the translation. could be improved by taking larger training sets of paired sentences. As I previously noted. the entire corpus of the European Parliament is available as a training set if one wanted to use it. The translation of individual sentences, even those which are part of the training set, will not necessarily be accurate , due to the statistical nature of the process. The actual translations offered by google and the coming handheld devices will no doubt be much better than that achieved with the tutorial example I followed.