CSC 244/444 Nov. 22, 2021

Neural Networks[1]

Neural networks have proved very effective for learning to classify images, speech, text and other patterns, but also to transduce such patterns into other patterns, for example mapping one language to another (or to a logical form), mapping images to captions, generating possible continuations of given partial stories, etc. However, pattern transduction falls short of thinking, and neural networks require massive training data, whereas people learn concepts from far fewer examples, and learn new facts by being told or reading.

1 Evolution of Neural Net Technology towards "Deep Learning"

What are neural nets?

(Artificial) Neural Networks (ANNs, or just NNs), as understood in AI, are networks of computational units that resemble networks of neurons in the brain. I.e., the units compute a weighted sum of their inputs, and apply a "threshold function" (also called an "activation function") to produce a single output. This output is 0 or low if the weighted sum of inputs is below threshold, while it is high (typically close to 1) if the weighted sum of inputs is above threshold. The weights applied to the inputs received by a unit can be altered in the course of learning. They can be thought of either as (modifiable) properties of the unit receiving the inputs, or of the unit-to-unit connections.

Each neuron-like unit has a number of input connections to it and a number of output connections leading away from it. Typically, units are arranged into layers, and function either as *input units*, *hidden units*, or *output units*. Input units have a single input or multiple inputs (which could be discrete or continuous) supplied from the outside, where each input represents an element of the pattern to be processed; input units (typically) transmit their output to multiple hidden units in the next layer via connections to those units. Hidden units receive inputs from the previous layer, and transmit their output through multiple connections to the next layer. Output units receive their inputs from the previous layer, and their output represents an element of the NN's final output. For simple binary classification tasks, there might be just one output unit, supplying a choice between the two classes. For multiple-choice tasks, we might have multiple outputs each of which gives the probability of one of the choices being the correct one, with these probabilities adding up to 1.

[1] Caveat: This an overview by a non-expert on this topic; For an up-to-date reference see Jurafsky & Martin's *Speech and Language Processing*, ch. 7, http://web.stanford.edu/~jurafsky/slp3/7.pdf

A note about counting layers: There can be some ambiguity about how many layers a NN has. Suppose that each unit of the initial "layer" sees the value of just one element of the input pattern and neither applies a weight to it nor does any thresholding; i.e., it simply transmits the given value to some number of units in the next layer. But then that next layer is not hidden – its inputs are known; so we may well discount the layer of transmission points as a layer of NN units. If, on the other hand, the input units do weight and threshold their inputs, then even if each input unit just sees one element of the input pattern and transmits its output to just one successor unit, it would be counted as an input layer. In general, we can properly count any layer of units that perform computations (other than identity) on their inputs as NN layers, even if the weights applied to inputs are not adjustable, or no thresholding is done.

Some types of NNs

Perceptrons. The simplest kind of NN, a *perceptron*, has just one unit, accepting some set of inputs and thresholding their weighted sum. Perceptrons were shown to be able to learn to classify inputs (e.g., handwritten characters) by training them on labeled examples. Training involves use of classification successes and errors to modify the weights of the connections from the inputs to the thresholding function, "rewarding" those connections that contributed to correct outputs and "punishing" those that contributed to incorrect outputs.

Perceptrons can learn to compute classifications only where the classes of inputs are *linearly separable*. For example, in two dimensions (say, 2 input units with continuous inputs), this implies straight-line separability of clusters of points in the Cartesian plane corresponding to distinct classes; in three dimensions, it implies separability by a plane, and in higher dimensions it implies separability by hyperplanes. Examples of classification problems unsolvable by perceptrons are input parity (whether there is an even or odd number of binary inputs) and connectedness (whether or not the "black" pixels in a square binary array of black and white pixels are connected pixel-to-pixel or not). The discovery of these limitation by Minsky & Papert in the late 1960s put a damper on the initial hype that had accompanied the prior literature on perceptrons. Remarkably, however, it was shown in later work that NNs with just one hidden layer (thus, 3 layers) could in principle compute any boolean function, could approximate any bounded, continuous function, and implement any decision boundary; and with two hidden layers, any function could be approximated. However, these results are not necessarily practical, because an exponential number of hidden units may be required for a given level of approximation. In practice, adding more layers (possibly dozens) has been found to improve NN performance. Also, note that function computation is intuitively inadequate for reasoning over a knowledge base, which may involve a number of steps with no fixed upper limit, and the knowledge base may be continually added to.

Feed-foward NNs. Networks in which signal propagation is unidirectional from layer to layer (in effect, an acyclic directed graph) are called *feed-forward* NNs. There's a

very convenient mathematical view we can take of such networks, where we regard each hidden layer L as performing a matrix-vector multiplication, adding a vector of bias amounts, and then thresholding the resulting vector uniformly (assuming that all units in the layer use the same threshold function).

Why matrix-vector multiplication? Just think of each unit of layer L as receiving as its input the *entire vector of outputs* of the previous layer. We can think of it that way, because if a connection from a unit j in the previous layer to a unit i in layer L is missing, this is equivalent to unit i assigning zero weight to the output of j. Now imagine the weights that are applied by the units in layer L as rows of a matrix; i.e., row 1 contains the vector of weights that unit 1 applies to the vector computed by the previous layer (and thus supplied as input to layer L), row 2 contains the vector of weights that unit 1 applies to the vector computed by the previous layer, and so on. So you can see that the product of this matrix times the vector of values received from the previous layer yields another vector, consisting of the dot products of the rows with the received vector: the desired linear combination of the values received from the previous layer. This dot product is then shifted by some bias amount by the added bias vector, and the result is thresholded to produce a component of the output vector of layer L. We'll reiterate the above perspective a little more formally later. Given this matrix-vector based view of what an NN layer does, it is often natural to represent such a layer as a single node or complex unit, performing a vector-to-vector transformation, i.e., a linear transformation followed by uniform thresholding.

Recurrent NNs and LSTMs. Networks allowing cyclic signal pathways are called *recurrent* networks. Important kinds of recurrent networks include Hopfield nets and Boltzman machines, in which connections are bidirectional (and thus certainly "loopy"). In Boltzman machines activation of units is probabilistic, rather than rigidly determined by the inputs. Such networks can model associative access, in the sense of approximately reproducing a complete learned pattern based on inputting parts of the pattern. In effect they sparsely encode the learned patterns in the hidden layer(s).

In many natural language processing applications, recurrent neural nets with loops serving as memory, particularly LSTMs (Long Short-Term Memory neural networks) have played a key role. LSTM "modules" are in general made up out of units that transform vectors, i.e., the inputs are vectors and the outputs are vectors (as per the matrix-vector view described above). In LSTMs, a central (complex) hidden unit computes its output vector based on "seeing" not only its current input vector (such as a concatenation of "word embeddings" – words coded as vectors in a way that puts words occurring in similar contexts close together in vector space) but also its own previous output, modulated by an "input gate". Roughly speaking, the gate may at various times emphasize or de-emphasize the previous output of the central unit (depending on the input), and this corresponds to the central unit "remembering" an earlier output, or instead paying more attention to the current input in computing an output. The gate works by applying weights to the vector components, but the weights are themselves

altered at every step as a function of the inputs. An LSTM also has an "output gate", that decides how much of the central unit's output to pass on.

Transformer models The sequence-to-sequence style of text processing by LSTMs has more recently (since about 2017) been overtaken in various applications by *transformer models*. These use a so-called *attention mechanism* to decide what (derived) features of an input sequence to focus on in deriving outputs, rather than processing inputs sequentially and trying to "remember" important input features detected earlier. They are usually divided into two main parts, an *encoder* and a *decoder*, each consisting of multiple NN layers. In early versions of the attention mechanism (typically added to a sequence-to-sequence model) the idea was just to dynamically identify segments of the final encoder output vector (e.g., segments corresponding to words, or rather to feature vectors derived from the embeddings of the words) that were particularly relevant to computing the desired output (e.g., in next-word prediction). This identification is done with attention subnetworks that pick out certain segments of the output of the final decoder layer as particularly relevant to extending the decoder output, communicating this information to the decoder.

Later versions abandoned the recurrent aspects of encoders and decoders, instead using attention more extensively. In particular, several layers in both the encoder and the decoder use "self-attention": For each output segment x_i of such a layer, matrix operations are used to determine its "relevance" to each other segment x_j. (Essentially, feature correlations are detected; intuitively, for example, features of the subject of a sentence may be found to be relevant to features of the verb of the sentence.) these i-j relevance scores at segment x_i are combined with a linear transform of x_i itself and the result is passed on to the next layer in the encoder or decoder. In the attention-modulated layers of the decoder, not only self-attention is used to determine outputs to the next layer, but also encodings generated by the encoder layers. The layers with self-attention in both the encoder and the decoder are followed by several more standard feed-forward layers. The final decoder layer usually generates a distribution over likely answers (using softmax – exponentiated individual answer values divided by the sum of all exponentiated answer values).

Typically transformer models are pretrained on very large amounts of general data, and for particular kinds of tasks are "fine-tuned" on examples of those tasks, altering the weights in just the last few layers of the encoder and decoder. Language models based on transformers (most famously GPT-3, and also the less data-huungry GPT-2, BERT, and RoBERTa) can be used for word or sentence prediction based on quite long (multi-sentence) prior text segments, for filling in missing words or phrases, summarizing, answering questions, engaging in chat, etc. They are extremely good "mimics", when trained on very large amounts of data; for example, GPT-3 was trained on 45 terabytes (45 million million bytes) of textual materials (Wikipedia, novels and other books, news media, social media like Twitter, WeChat, Quora, etc.) and uses 175 billion parameters. However, such machines do not think or reason, based on provided information in

conjunction with background knowledge, as people do.

Types of learning methods in NNs

In the 1960s and 70s, an obstacle to effective use of multilayer NNs was a lack of good methods for training such networks using labeled examples: How can one assign "credit" or "blame" to connection strengths of inputs to hidden units? This problem was solved by using *differentiable* threshold functions and *backpropagation*. Essentially, this involves taking the derivative of the (squared) output error (for a given NN input pattern) with respect to each connection weight in the NN, and adjusting the weight in a positive or negative direction depending on whether that derivative is negative or positive respectively. In the original conception of perceptrons, threshold functions were discontinuous step functions, and as such not differentiable. It was the introduction of continuous *sigmoid* functions or *hyperbolic tangent* functions as threshold functions (see below) that made backpropagation possible. The mathematics of backpropagation was already developed in the 1960s, but it was not until the 1970s that the applicability to NN learning began to be appreciated, and it was only in 1986 that an experimental demonstration of the technique by Rumelhart, Hinton, and Williams firmly established the utility of the method.

Even with backpropagation, successes in training multilayer NNs were limited by the need for huge numbers of training examples, and by the tendency of iterative learning to settle into local minima, instead of finding weights that globally minimize errors. NN developers generally had to be quite clever in engineering good "features" of an input, for a given classification task, that would ease the learning problem for the NN. This detracted from the idea of NNs as a means of learning arbitrary classification tasks. A technique apparently due to Bourlard and Kamp (1988) was "auto-association" or "autoencoding". A basic autoencoder has at least an input layer, called an encoder (think of it as having a large number of units, allowing for complex input patterns), a hidden layer with fewer units (think of it as representing the input in an efficient "code"), and an output layer, called the decoder (with the same number of units as the input layer), which is intended to reproduce the input as nearly as possible. To learn to reproduce its inputs (based on numerous input examples), the system is forced to learn an efficient encoding (one of "lower dimensionality") of its inputs. The important point is that these efficient encodings are learned in a *unsupervised* manner, i.e., the inputs needn't be labeled with desired outputs, because the desired outputs *are* the inputs! The encoding/decoding idea is key to many other tasks. For example, in language modeling a NN may learn to encode fixed-length segments of text in such a way as to optimize its prediction of the following word or words. In machine translation, an NN learns to encode source language sentences in such a way as to optimize generation of the corresponding sentence in the target language. (Of course the latter is an example of supervised learning – we need to have the example translations from which to learn.)

In 2006, Hinton *et al.* showed how to make the autoencoding technique practical.

In effect they trained layers of restricted Boltzman machines so that each layer (with fewer units than the previous layer) learns the inputs to the previous layer (thus the first hidden layer learns the external inputs, and its outputs serve as inputs to be learned by the next, sparser, layer, etc.). The final (output) layer consists of units fully connected to all units of the previous layer, and only the last one or two layers of the NN are trained in supervised fashion, i.e., the NN seeks to obtain the given, correct outputs for a set of training inputs. In this training phase, the weights of the previously trained "autoencoder" levels are held fixed. In effect this kind of network discovers for itself what features of an input to derive in order to be able to compactly encode the inputs; the successive levels learn more and more abstract features, and these generally serve well for miscellaneous learning tasks, as inputs to the last layer or two.

This work again caused much excitement, and the term "deep learning", was increasingly used in the literature about such multilayer NNs. Ultimately, it turned out that the availability of web-scale training data made Hinton *et al.*'s autoencoder approach unnecessary for learning from "big data" (though not for modest-size datasets). For example, a multi-layer feed-forward network trained in supervised fashion on many millions of images labeled with objects occurring in those images could be used almost as-is for many other tasks. One just needed to retrain the last couple of layers on labeled examples for the task at hand (e.g., skeletal joint positions on people in an image – as recently implemented by a local grad student, Iftekar Tanveer). Interestingly, in an NN trained on "big data" for a sufficiently diverse task the earlier hidden layers automatically learn to encode the inputs efficiently in a more or less task-independent way, at successive greater levels of abstraction, just as if autoencoding had been used for training.

Convolutional neural networks

It should be added that the successes of the above deep-learning systems have also been aided by a particular way the earlier layers are organized: Each of several initial hidden layers consist of identical units with identical input weights, each taking inputs from a small (say, 9-element) local "patch" of the input (where patches slightly overlap). Computing the weighted sum of the elements of each input patch, performed uniformly over identical, slightly overlapping patches that "tile" (cover) the entire input pattern, amounts to mathematical *convolution*. Another way to think about it is as matching a template to each patch of the input, and measuring the quality of the match; the match results are then the outputs of the convolutional layer. The use of convolutional templates was inspired by the "receptive fields" that play a key role in mammalian (including human) vision. Often the convolutional layers are followed by "pooling" layers, which again take their inputs from "patches" of the outputs of a convolutional layer, but in this case they use adjacent, nonoverlapping patches and each computes the maximum value in a patch. This reduces the dimensionality of the pattern being processed by a factor equal to the number of values within a patch from which the maximum is chosen. The effect is to ensure a degree of insensitivity to exactly where the convolution values

(template matches) were highest in a local region.[2]

The uniformity of the convolutional and pooling layers in convolutional NNs (CNNs) significantly eases the learning problem, since only one set of weights, used by each unit in a convolutional layer, needs to be learned for such layers, and pooling layers implement fixed max-functions. CNNs have been successfully applied to image and video processing, various NLP tasks (speech recognition, query-based document retrieval, syntactic and semantic parsing, machine translation, etc.), drug discovery, and the game of Go, in particular Google's AlphaGo (this defeated a 9-dan Go player 4 games to 1; the program also used special move prediction networks, board-value networks, and Monte Carlo game-tree search); the later AlphaGo Zero program was even stronger, playing well above human levels. (Coming back to large transformer models like GPT-3 for a moment, it's interesting to note that these have also proved capable of producing results similar to those produced by CNNs; they have also proved capable of playing chess – which after all can be viewed as a transduction from chess board configurations to moves.)

These recent successes (along with the "Watson" Jeopardy win and many news items about self-driving cars) have led to much hype about imminent human-level AI. However, no NNs so far understand language, or learn to reason, plan, or engage in dialogue the way people do. They are still totally dependent on huge amounts of data for pre-training, plus fairly extensive supervised training for particular tasks in order to learn those tasks, whereas people learn concepts from relatively few examples and can learn facts in a single "shot" from another person or text source. Currently the most advanced systems for reasoning, planning, or engaging in dialogue are still ones based on symbolic representations, rather than NNs.

2 Some Specifics

The behavior of individual units

As noted, standard NN units are thought of as "threshold units" because, in analogy with organic neurons, their output may be close to 0 until the net value of the inputs reaches a certain threshold, and will then rather abruptly become much higher. More precisely, suppose that unit i receives inputs from units $1, 2, ..., n$; (normally unit i itself won't be one of these, unless it feeds its own output back to itself as an input). Suppose further that the outputs of those units are $s_1, s_2, ..., s_n$ respectively; (below we'll write them as $s_{i1}, s_{i2}, ..., s_{in}$ to make clear that units $1, 2, ..., n$ are the ones with connections to unit i; the point is that each unit in general receives inputs from a distinct set of prior units). The connections from units $1, 2, ..., n$ that transmit their outputs to unit i are assumed to have *weights* (connection strengths) $w_{i1}, w_{i2}, ..., w_{in}$. These can be positive or negative, and we can think of a signal transmitted over a connection with a positive weight as "excitatory" (it tends to make the target unit "fire") and one transmitted over

[2]For a max operation the derivative still exists, though it is a step function, i.e., the second derivative has singularities

a connection with negative weight as "inhibitory" (it tends to prevent the target unit from firing). The net input net_i to unit i is taken to be the weighted sum of the signals received from units $1, 2, ..., n$ (i.e., the *dot product* of the weight vector and the signal vector) plus a bias constant b_i:

$$net_i = \sum_{j=1}^{n} w_{ij} s_{ij} + b_i.$$

This is the value that feeds into the threshold function (activation function) f of unit i, causing the unit to produce a 0 output or very low output when net_i is below the fixed threshold, and an output near 1 when net_i is above threshold:

$$s_i = f(net_i), \quad \text{where, e.g.,}$$

$$f(x) = \frac{1}{1 + e^{-x}} \quad \text{(a sigmoid function).}$$

You can see that for very negative x, $f(x)$ is close to 0, for $x = 0$ it is 0.5, and for $x > 0$, it rises to 1. Thus 0 can be thought of as the threshold value (but the threshold is really "smeared out" around 0). There are other sigmoid functions besides this so-called "logistic function", i.e., differentiable functions with this sort of S-shape. An advantage of the logistic version, is that its derivative is expressible in terms of itself:

$$f'(x) = f(x)(1 - f(x)).$$

This makes the derivation of the backpropagation rules easier. Other popular threshold functions are $f(x) = tanh(x)$, which rises from -1 to $+1$ (with value 0 at $x = 0$) and whose derivative is $(1 + f(x)(1 - f(x))$; and the *rectified linear unit* (ReLU) that is 0 for negative x and rises up linearly from 0 – thus its derivative is simply 0 for negative x and constant for positive x.

Finally an important point to be reminded of once again is that we can view the transformation performed by a NN layer as a matrix-vector multiplication followed by adding a bias vector and thresholding the resultant vector. In terms of the notation above, we would assume that every unit i of an m-unit NN layer receives the entire set of signals $\vec{s_i} = s_{i1}, ..., s_{in}$ from the previous layer, i.e., $\vec{s_i}$ is actually the same for all i. Each unit forms a dot product $\vec{w_i} \cdot \vec{s_i}$, adds a bias term b_i, and applies thresholding function f to this. Again, if every unit i does this, the result is a vector of m thresholded, bias-shifted dot products (one component for each NN unit) – and that is the same as saying that we are applying an $m \times n$ matrix W of weights (where each of the m rows of length n corresponds to a NN unit) to the signal vector $\vec{s_i}$, adding a bias vector $\vec{b_i}$, and then using f to threshold the elements of the resulting m-vector. In short, the NN layer forms $f(W\vec{s_i} + \vec{b_i})$. As noted earlier, the assumption that $\vec{s_i}$ is independent of whichever unit i we're considering is unimportant, because a unit can "select" which inputs to take into account by applying weight 0 to the rest. In actual training of a network we might start with full connections and non-zero weights, but eventually zero-out components that seem not to affect the quality of the outputs. (In brain development from infancy,

synaptic connections become very dense at first, and later are "pruned" to become more sparse.)

Backpropagation

First you should note that since we have a mathematical expression for the input-output behavior of each unit of an NN, we can also derive a mathematical expression for the output of each unit (including the NN output units) of the NN for any given input. Thus we can conceptualize the problem of finding the optimal weights w_{ij} and biases b_i, given a set of training examples, as a problem of minimizing the average output error over all the examples. This is a mathematical problem that can be tackled, for example, by gradient descent methods, since our functions are all differentiable.

However, from a learning perspective it is more convenient to use a method that learns example-by-example, and that is what backpropagation enables. We won't go through that derivation (which is based on the chain rule of differentiation, whose general form is $\delta f(g(...))/\delta w = f'(g(...))\delta g(...)/\delta w$, where w is some parameter whose influence on $f(g(...))$ we are trying to determine). Backpropagation makes weight adjustments in proportion to the derivatives of output error terms at each layer, working backward from the final layer. The secret of success lies in using the right notion of an "error term" at each layer, enabling the recursive computation of the derivatives of the net output error with respect to each weight, and hence the direction of adjustment of those weights. The algorithm is simple enough to be stated here for a feed-forward network:

Consider a particular training example;

1. Compute the output s_i for each unit i, in a forward sweep from the input layer to the output layer;

2. For each output unit i, compute its *error term*,
   $\delta_i \leftarrow s_i(1 - s_i)(t_i - s_i)$,
   where t_i is the correct (desired) output value at output unit i. Note the resemblance of $s_i(1-s_i)$ to the derivative of the logistic sigmoid function; and of course $(t_i - s_i)$ is the error at output unit i;

3. For each layer whose successor layer has already been processed:
   For each unit i in the layer, compute its error term,
   $\delta_i \leftarrow s_i(1 - s_i) \sum_{k \in outputs} (w_{ki}\delta_k)$;
   note that w_{ki} is the weight of the connection *from* unit i to successor unit k – whose error term δ_k we have already computed;

4. Increment each network weight
   $w_{ij} \leftarrow w_{ij} + \eta\delta_i s_{ij}$,
   where η is a constant called the *learning rate* (which should be small enough not to give excessive importance to a single input-output example); recall that s_{ij} is

the output of unit j, but indicating that it is connected to unit i; (we could have just written s_j).

We can't expect to get an optimal set of weights after a single pass through the training set; rather, we iterate until weights no longer change significantly. Often a *momentum term* is added to the increment in step 4, which is just a constant ≤ 1 times the increment added at the previous iteration. This can accelerate convergence and help avoid local minima.

As noted earlier, in applications to language modeling and other "sequence modeling" tasks, *recurrent* NNs (RNNs), such as LSTMs have often been used, which contain feedback loops that enable "remembering" and using features of input segments seen earlier in the processing. As also noted, more recently transformer models have begun to dominate. We have often referred to natural language processing, since it seems natural to try to redeploy these NLP techniques to reasoning tasks; after all, logic is ultimately derivative from propositions and argumentation expressed in ordinary language. However, NNs are by their nature one-step, direct transducers from one pattern to another, rather than mechanisms that retrieve information from an explicit propositional memory and combine the retrieved information in as many steps as required to reach a relevant conclusion. So progress towards NN-based understanding and reasoning is handicapped from the outset, and the results that exist generally come down to one-shot inference by analogy (i.e., mimicry of training examples).