Neural Networks

CSCI 2400 "Part 10"
Instructor: David Byrd

Disclaimer: Lecture notes can't and won't cover everything I say in class. You should attend class each day and use these for review or reinforcement.

In this set of notes, we first delve into the detailed operation of a single neural computation unit, then consider the potential of a connected computational graph of the same.

12 The Artificial Neuron

The learning construct known as an Artificial Neural Network (ANN) was inspired by the human brain's own massively-parallel "network" of very simple computational units – neurons! Since their initial conception, it has become more and more apparent that ANNs do not really reflect the actual functioning of a biological brain, but they are still quite useful as a mathematical tool.

A key benefit to ANNs is that in theory they can learn to approximate *any* function. Drawbacks include a requirement for vast amounts of training data, very slow training speed, and a demonstrated performance weakness on certain types of complex, discrete problems, such as those for which a decision tree might be optimal.

12.1 Input to an Artificial Neuron

Individual artificial neurons (which I will henceforth just call neurons) will typically be organized into a network of interconnected neurons. We will get into the structures of such a network later, but for now simply think of a large number of neurons all connected together in some complicated manner.

A single neuron can have many inputs, each of which arrives from the output of some other neuron in the network. At any given time, these input connections will be "activated" with some signal strength, usually between -1 and 1, depending on how strongly the previous neurons were excited by their own inputs. For now, the output of the neuron will be the sum of the inputs.

We label the output activation strength of each neuron as a_i, where i is the neuron's arbitrary but unique identifier in the network. In this example, we are inspecting neuron 4, which has inputs from neurons 1, 2, and 3.

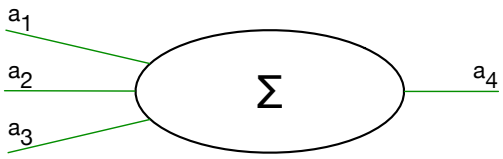

Figure 1: A simplified artificial neuron.

The current neuron model is not very useful. Numbers can flow around in such a network, but what learning can take place? The output of our example neuron is simply $a_4 = a_1 + a_2 + a_3$, and our neuron has no "control" over its output value. There are thus *no degrees of freedom* in this model (no parameters that can be altered to improve performance) and learning cannot take place.

We can correct this defect by attaching a *weight parameter* to each input connection of the neuron. This weight is a real-valued number that controls how strongly this neuron "listens to" the neuron connected to this input. The input activation signal and the corresponding weight are multiplied together. Thus, a weight of +1 means the neuron takes the input activation strength at full value (whatever it is). A weight of 0 means the neuron ignores this input value. A weight of -1 means the neuron takes the *opposite* (negation) of the input from the previous neuron.

Now we can improve our neuron model to look like this:

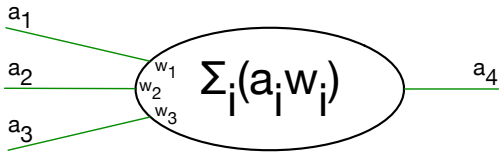

Figure 2: A better artificial neuron (one that can learn).

Our neuron now computes something that should remind you of *multivariate linear regression* from the Local Search part of the course. (We used it as the example learning application for gradient descent.) In multivariate linear regression, we wish to learn the parameters $m_{0\ldots n}$ that will best fit the input $(X_{1\ldots n}, y)$ training data with minimum loss, using the function:

$$y = m_0 + m_1 x_1 + m_2 x_2 + \cdots + m_n x_n$$

In fact, this *is* multivariate linear regression! The $x_1 \ldots x_n$ inputs are our input activation strengths $a_1 \ldots a_n$, and the learned parameters $m_1 \ldots m_n$ are our learned input weights $w_1 \ldots w_n$. We are given inputs and a target output $(a_{1\ldots 3}, a_4)$ and we try to learn the parameters $w_{1\ldots n}$ that will best fit the training data with minimum loss, using the function:

$$a_4 = w_1 a_1 + w_2 a_2 + w_3 a_3$$

Where is m_0? If you recall, we decided to add an intercept parameter (one not tied to any x input) to our linear regression learner. Without such a parameter, it could only represent

hyperplanes that passed through the origin. With the m_0 "translation" parameter, it could represent *any* hyperplane.

In fact, it turns out to be useful to include such a parameter in the neuron as well. We will call it w_0 and tie it to a constant input a_0, the "bias" input. The bias input a_0 is typically -1 exactly and is offered to all neurons in the network. Each neuron has its own w_0 weight to control how much it "listens" to the bias input. It will become clear why we choose $a_0 = -1$ in the neuron output section, below.

With this improvement, we now have our complete input model for a neuron, and we can now perform *multivariate linear regression* inside every individual neuron in the network. (Do you start to see why this is such a powerful learning technique?)

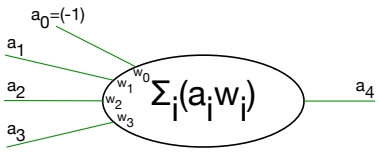

Figure 3: The final input section for a neuron, including bias.

At this point, our neuron can compute multivariate linear regression to fit the following function:

$$a_4 = -w_0 + w_1 a_1 + w_2 a_2 + \cdots + w_n a_n$$

If we simply output a_4 under the current model, we will create a problem. While $a_{1...3}$ were in the range -1 to +1, the weights $w_{1...3}$ are not required to be, and the weighted sum a_4 is *likely* not to be. When the following neurons take $a_4 > 1$ as an input and add it to other inputs, the activation strength will grow even larger. We will quickly have "exploding" activation strengths in our network. We would prefer to keep the activation strength in the same range of -1 to +1 at each layer. Let's consider improving this situation on the "output" side of our neuron.

12.2 Output of an Artificial Neuron

In the previous section, we decided it was not a good idea to simply output the raw weighted sum of our inputs as a_4. Given that, we will need a new name for the weighted sum of the inputs $\sum_i a_i w_i$. Let's just call it *in*. One problem we are trying to solve is having the activation strengths grow without bound as we move further "right" in a network (due to being the sum of sums of sums of sums, etc).

The easiest way to constrain our output would be to make it binary, with some *threshold* for *in* at which the output will suddenly flip from off (zero) to on (one). For simplicity, we will set that threshold at zero. Now that we are constructing an *output function* for our neuron, we should also give it a name. We will use g and call it the *activation function*, because it maps the neuron's input to some activation strength for the neuron's output.

So for our neuron, we now have:

$$in = \sum_{i=0}^{n} in_i = \sum_{i=0}^{n} w_i a_i$$

$$g(in) = \begin{cases} 1 & \text{if } in > 0 \\ 0 & \text{if } in \leq 0 \end{cases}$$

$$out = g(in)$$

Visually, it looks like this:

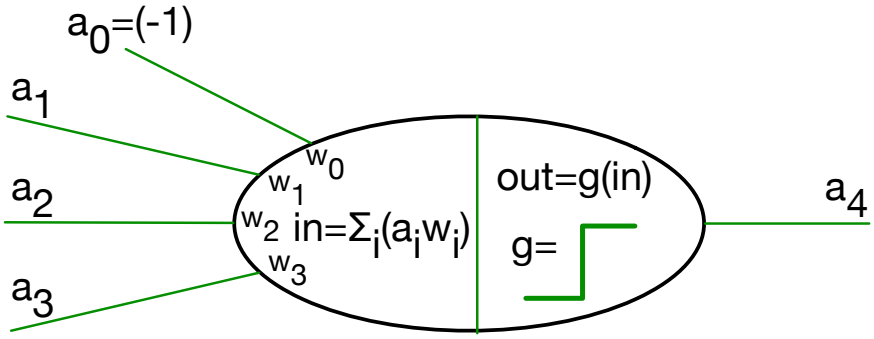

Figure 4: Neuron with a threshold activation function.

We can now clearly see the purpose of the $a_0 = -1$ bias input to each neuron. The activation threshold for $g(in)$ is locked at 0. What if our neuron has the correct *relative* weights for its inputs, but "learns" that it should require a higher total level of activation before it fires its output? Since we cannot directly alter the threshold for g, without the bias input, the neuron would have to carefully change *all* of its input weights downward while trying to keep them in the same proportion to one another. Instead, we can simply increase w_0, the weight that controls the bias input.

It is helpful to think of a few specific examples. The bias input $a_0 = -1$ always. So when $w_0 = 0.5$, the neuron is adding $w_0 a_0 = 0.5(-1) = -0.5$ to its total input. When $w_0 = 0$, it adds 0, effectively ignoring the bias input. If $w_0 = -0.5$, the neuron adds $w_0 a_0 = -0.5(-1) = +0.5$ to its total input. *This is the same as adjusting the threshold of the activation function.*

12.2.1 The Perceptron and its limits

A neuron with this simple threshold activation function is traditionally known as a *perceptron*. What can we do with a single perceptron? One interesting thing is to turn it into a logic gate. A single perceptron can learn to behave exactly as an `AND`, `OR`, or `NOT` gate (or boolean logic function). Because it represents a piecewise step function with a single threshold, however, it cannot represent the `XOR` function. For `XOR`, as the total input to the perceptron increased, it would need to be able to be not firing (both inputs off), then firing (one input on), then not firing again (both inputs on). This is not possible for our simple perceptron.

A connected network of simple perceptrons, then, should be able to represent any statement of boolean logic. Our textbook has a nice image showing how a perceptron can represent different boolean logic functions, if you missed my drawing in the lecture.

There are at least two problems with this simple perceptron. One, even if we combine very many of them, it will be hard to represent an arbitrary nonlinear function, because each perceptron's output is a "hard edged" step from zero to one. It would require infinitely many to produce the "smooth" surface of some curving nonlinear function. Two, as it turns out, we will use gradient descent (see the Local Search notes) to train our networks, and the derivative of our threshold activation is either zero or *undefined* at every input. That isn't going to work – there's no gradient to descend!

12.2.2 The Activation Function

Except for the unbounded growth of our activation strengths, we had the best possible activation function the first time – the identity function $out = in$. As much as possible, we would like to retain this behavior: an activation function that usually lets our input pass through unmolested, but compresses it when the value becomes too large or small.

For these reasons, a more common activation function choice in modern neural networks is something from the *sigmoid* family. These functions have slope near 1 when close to the origin, tapering off to slope near 0 at the extremes, thus "capping" our output as we require.

The sigmoid function itself ranges from 0 to 1 and is simply $1/(1+e^{-x})$. This produces an S-shaped curve that *approximates* a threshold at $in = 0$, but transitions from 0 to 1 smoothly instead of suddenly. Thus it solves the two problems we had: it can be used to create smoothly-sloping surfaces for our overall function approximation, and it is differentiable everywhere with a nice gradient to follow. The sigmoid function is known by two other common names: the *logistic* function and the *soft threshold* (or *soft step*) function. These are all the same thing.

A drawback to the sigmoid function is that it does not pass through the origin. At zero in, its output value is 0.5, and thus it does not approximate the identity function ($out = in$). A better choice could be the tanh function, which has similar shape, but ranges from -1 to 1, centered on zero, and *does* pass through the origin. (Also popular right now are the *ReLU* or Rectified Linear Unit, which is the identity function when $in > 0$, but a flat line of $out = 0$ when $in < 0$, and the *Leaky ReLU*, which approximates the previous but is not a perfectly flat line for $in < 0$.)

13 A Network of Neurons

The primary neural network construct that we consider in this class is the Feedforward Neural Network (FFNN), which is a network of neurons with connections only in one direction (no loops). It is typically *fully connected*. What do we mean by fully connected? Imagine a FFNN laid out in parallel layers. Think of each layer as a vertical column of neurons, with multiple columns flowing from left to right. A neuron is connected to all of the neurons in the layer before it (left) and after it (right), but *not* to the other neurons in its own layer. Computation flows through the network from left to right.

13.1 Neural Network Connectivity

The diagram below may help to illustrate the connections in a FFNN. For a single estimation of y from X, we pass the X values to the input layer (left) of the neural network, one per neuron. Each layer's output activation strengths form the inputs to the next layer to the right. The output activation strengths of the output layer (right) are interpreted as the network estimates for the y values. Any layer that is neither the input nor output layer is called a *hidden* layer, because their (actual or desired) activation strengths do not appear in the training data. They simply do *whatever is necessary* to help produce the correct output.

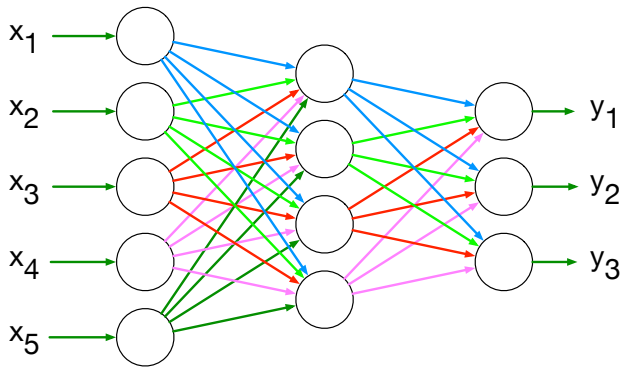

Figure 5: A simple feedforward neural network encoding the function: $f(X) \to Y$.

In the diagram, the color coding is intended only to make it obvious that the network is *fully connected*. That is, each neuron is connected to *all* neurons in the layer before and after it. Note the lack of connections within a layer. Each neuron also has a constant bias input, omitted from the diagram to reduce clutter.

A feedforward neural network, by definition, contains no cycles or loops in the calculation pathway. One input vector is fully pushed through the network and the estimate is obtained from the output layer before another input is offered. Thus we can say that a FFNN *has no internal state* other than the neuron weight parameters. During training, these weights will change to learn the desired function, but are then fixed when the network is simply being used to perform calculation. Thus in a production environment, the FFNN has no ability to "remember" anything for later use. To put it another way, *a FFNN represents a function of its current inputs*.

As shown above, it is entirely reasonable for a FFNN to have multiple outputs. This means it represents *multiple* functions simultaneously: $f_1(X) \to y_1, f_2(X) \to y_2, f_3(X) \to y_3$. If the network contains a hidden layer, then there are some *shared parameters* (weights) among these functions, so they are not fully independent of one another. If there is no hidden layer, there are no shared weights and the functions *are* independent. (If this is not clear, look at the diagram again. In the top neuron of the hidden layer, the input weight from the top input neuron affects the values of y_1, y_2, and y_3.)

13.2 Neural Network Example

For a small, specific example, we can actually write down the exact function being learned by the network. Doing so can help illuminate what a neural network really is, and why we

call one a *continuous function approximator*. Look at the following diagram:

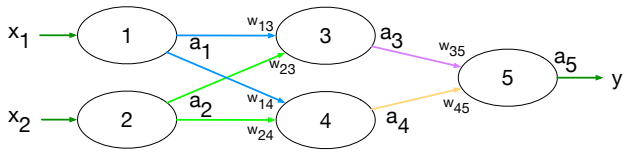

Figure 6: A small, fully-detailed feedforward neural network.

Above is a small feedforward neural network that maps some simple function: $f(x_1, x_2) \rightarrow y$. The input layer of neurons 1 and 2 is drawn for clarity, but note that in most implementations they are not "real" neurons. The input layer is simply a mechanism for providing the input X values into the network. There are thus three neurons that together calculate the output estimate for this network: neurons 3, 4, and 5. Each neuron i has a bias input of -1 as usual (omitted from the diagram), and a weight b_i which controls its use of that bias input. (These are sometimes labelled as w_{0i}, with the bias input as a_0.)

Recall that the function computed by a single neuron (refer to section one of these notes) is $out = g(in)$ using some activation function g. Starting with the observation that $a_5 = y_{pred}$ (we interpret a_5 as the network's estimate of y), we can then expand the exact parameterized function represented by this network:

$$\begin{aligned} a_5 &= g_5(-b_5 + w_{35}a_3 + w_{45}a_4) \\ &= g_5\big(- b_5 + w_{35} \cdot g_3(-b_3 + w_{13}a_1 + w_{23}a_2) \\ &\qquad +w_{45} \cdot g_4(-b_4 + w_{14}a_1 + w_{24}a_4)\big) \end{aligned}$$

In the second line, we have simply expanded a_3 and a_4 in the same way we did for a_5. All of the weights (b and w) are parameters we will learn, and intermediate activation strengths a_3 and a_4 have been expanded away, so we can see that the function represented by the network is simply: $f(a_1, a_2) = a_5$. Since $a_1 = x_1$, $a_2 = x_2$, and $a_5 = y$, this is exactly what we expected.

Thus this $f(a_1, a_2)$ is the function represented by the network. If the output a_5 value is wrong, how can we change it? We are not permitted to change our training data, so a_1, a_2, a_5 are fixed. We must therefore improve the estimate by altering the bias and weight parameters: $b_3, b_4, b_5, w_{13}, w_{14}, w_{23}, w_{24}, w_{35}, w_{45}$. This is exactly how learning in a neural network occurs!

It is reasonable to think of the weights in the network as a large 2-D matrix with neurons i down the left side and j across the top. Then the value in position (i, j) of the matrix is w_{ij}, the weight applied to the activated connection between neurons i and j. This matrix now contains all the values we can change to improve the performance of our network, and so our goal is to search this "weight space" (all possible matrices) to find the best matrix that minimizes the error of our predictions. We can say our hypothesis or candidate solution is a particular weight matrix and our loss function tells us how good or bad this hypothesis is.

13.3 Training a Feedforward Neural Network

It is easy to see how we can train a single neuron, or even a single parallel layer of neurons, to produce the correct output values. We can use *gradient descent* to perform a local search of the weight space described above. We would take one training tuple (X, y), plug in the X values as inputs to the network, and calculate the y_{pred} output. Then we use y and y_{pred} to feed the loss function, calculate the partial derivative of loss with respect to the weights, and adjust the weights a little bit in the direction that shrinks the loss. This works well (as long as our activation function is differentiable) because one neuron is just calculating multivariate linear regression, which we have already seen how to train using gradient descent.

What if there are multiple layers? In the previous section's example network, we could train neuron 5 easily enough by plugging in $a_1 = x_1$ and $a_2 = x_2$, running the network to get $y_{pred} = a_5$, then calculating the loss function and its partial derivatives $\frac{dLoss}{db_5}$, $\frac{dLoss}{dw_{35}}$, and $\frac{dLoss}{dw_{45}}$. Using those values, we can see how to update b_5, w_{35}, and w_{45} to reduce the loss. The important thing to realize here is that this works *only* because we *know* what value a_5 was supposed to be (it should be y).

How about the layer before that, with neurons 3 and 4? Our method doesn't work, because we *don't know* what the values of a_3 and a_4 should be. Values a_1, a_2, a_5 are part of our training data, but a_3 and a_4 are not! How can we calculate the loss function if we don't know the "correct" answer?

We can't! We can, however, estimate it.

13.4 Backpropagation

The intuition of backpropagation is simple. Since we *can* compute the error at the output neuron(s), we will do that, then invent some method to propagate that error *backwards* through the network to obtain an *estimate* of the error at each hidden neuron.

How can we estimate the error at hidden layers? Each hidden neuron is responsible for *some portion* of the error at the following neuron. How much? It is proportional to the weight connecting the hidden neuron to the following neuron. That is, the more a subsequent neuron is "listening to" this neuron, the more of its error this neuron is responsible for, and the more this neuron should change to correct it.

We assume there are k output neurons and will use the sum of squared errors as our loss function: $Loss(w) = \sum_k (y_k - a_k)^2$. Then we can derive the necessary equation for the backpropagation algorithm as follows:

$$
\begin{aligned}
\frac{dLoss}{dw} &= \frac{d}{dw}\sum_k (y_k - a_k)^2 && (1)\\
&= \sum_k \frac{d}{dw}(y_k - a_k)^2 && (2)\\
&= \sum_k 2(y_k - a_k)\frac{d}{dw}(y_k - a_k) && (3)\\
&= \sum_k -2(y_k - a_k)\frac{d}{dw}a_k && (4)\\
&= \sum_k -2(y_k - a_k)\frac{d}{dw}g(in_k) && (5)\\
&= \sum_k -2(y_k - a_k)g'(in_k)\frac{d}{dw}in_k && (6)\\
&= \sum_k -2(y_k - a_k)g'(in_k)\frac{d}{dw}(\sum_j w_{jk}a_j) && (7)
\end{aligned}
$$

Equation (1) is simply our loss function of the sum of squared errors between a_k, the prediction of output neuron k, and y_k, the correct output value for neuron k. The derivative of a sum equals the sum of the derivatives, so in Equation (2) we pull the sum outside the derivative. In Equation (3), we have used substitution with $u = y_k - a_k$, so we now have only the simpler derivative $\frac{du}{dw}$. Equation (4) rests on the observation that *Loss* is a function of the network weights, but *not* the training values (X, y), which are fixed. The derivative of y_k (here, a constant) with respect to w will be zero. We also extract the -1 from $-a_k$ to the outside of the derivative. Equation (5) substitutes $a_k = g(in_k)$ from our basic definition of an artificial neuron. Equation (6) uses substitution again with $u = in_k$. Notation g' represents "the derivative of function g", as usual. Finally in Equation (7) we plug in $in_k = \sum_j w_{jk}a_j$ also from the basic definition of a neuron (a neuron's final input is a weighted sum of its input connections).

This backpropagation equation can be split into two pieces that suggest an algorithm. The first piece inside the sum (up to $\frac{d}{dw}$) represents the delta improvement for each neuron k in the current layer. The second piece (the $\frac{d}{dw}$ part) represents the neurons connected as inputs to k from the previous layer. These two pieces effectively give us an iterative way to pass a weighted portion of the error backward through the network, exactly as we desired.

The backpropagation algorithm can be outlined in pseudocode like this:

```
initialize all network weights to small, random numbers

loop:
    given (X,Y) training examples:
        run the network forward
        obtain predictions at each output neuron
```

```
    propagate the error (loss) backwards:
        for each node j in the output layer L:
            Δ_j = g'(in_j)(y_j − a_j)
        for each node i in each other layer from L-1 to 1:
            Δ_i = g'(in_i) Σ_j w_ij Δ_j
        for each weight in the entire network:
            w_ij = w_ij + (α a_i Δ_j)
until some stopping criterion is reached
```

As you read the algorithm, note that in the second and third steps of the propagation section, j is iterating over each neuron to which i is connected as an input, because these are the neurons *from which* the error will be flowing backwards to i. The delta variables are our estimated error at each neuron. Alpha represents a learning rate between 0 and 1, just as we used it for gradient descent in the Local Search section of the class.

Common stopping conditions are: error falls below some threshold, *change* in error between iterations falls below some threshold, test set or validation set error starts to increase, or sometimes simply a fixed number of iterations.

Hopefully, you can see the similarity between this and gradient descent, because backpropagation approximately *is* gradient descent that is using the chain rule for derivatives to work across the layers of the network. We require such a local search, because the weight space is *much* too large (infinite in the sense that it is continuous) to attempt to compute or search (in the A* sense) for an exact solution. Just as with gradient descent, we can construct a batch (all data used every loop), mini-batch (partial data each loop), or stochastic (one random data point each loop) version.

With enough training data, enough iterations, a suitable problem, and an appropriate network shape, backpropagation can converge to zero training error eventually. (Those were a lot of *important* qualifiers, though!)

As usual, we do not really want zero training error. (It is fine, but not what we care about.) We want *generalization*. The network should successfully produce accurate estimates for inputs on which it was *not* trained. So as with other learning techniques, we will use cross-validation, holding out some *test set* that is not used for training. The goal will be to eventually discover a setting of network parameters that performs best on the *test set*, not on the *training set*, in order to avoid overfitting the training data.

13.5 Shortcomings

There are some limitations and issues with backpropagation training, and with fully-connected feedforward neural networks generally, about which you should know.

1. They require *a lot* of training data.
2. They require labelled training data to compute the loss function. It can be difficult to find usefully-labelled training data in the volume required to train a neural network.
3. The time required to fit a network to a problem (i.e. the training time) scales badly as more hidden layers are added.

4. If the loss function is not convex (which might well be the case in some problems), the backpropagation algorithm can converge to a local optimum.

5. The design of the neural network structure itself (how many layers, how many neurons in each layer) is a problem for which there is as yet no clear solution. The subfield of architecture search considers this problem (i.e. "run Search to find the best size/shape/connectivity for the network").

6. Given many hidden layers to customize, a neural network can easily overfit simple problems by "memorizing" the training data.

7. For complex problems like computer vision to be solved by a simple FFNN, *extensive* feature engineering is required. We can't just throw raw pixels in as training inputs, but must design features that are "halfway there", like corners, edges, basic shapes, etc.

Deep learning addresses primarily the last issue, adding convolutional, recurrent, and other more exotic layers to the network, and more hidden layers in general. The theory that explains *exactly why* deep learning works so well on certain types of problems is not entirely clear (although there has been some progress). Nevertheless, in practice deep neural learning has become the state of the art in several fields of application. Deep CNN for computer vision does successfully turn raw pixels into correct answers *without* manual feature engineering. Deep RNN and its descendants, which add a self-loop to every neuron, to let its output be one of its own inputs at the next time step, has advanced language generation, parsing, and translation.

Due to shortcoming #4, these deep networks proved nearly impossible to train for years. When learning from random starting weights using backpropagation, they have a pronounced tendency to converge to local optima. This issue has been successfully addressed with layerwise pretraining, similar to a *stacked autoencoder*, which is beyond the scope of our class projects, but was discussed in lecture as a topic of potential interest. (Short version: starting from the input side of the network, pretrain each layer one at a time to reproduce its own input. This gives useful, instead of random, starting values for the final backprop training, and does produce better results on many problems.) Approaches like transfer learning (start with a network trained on a related problem and "fine tune" it) can now also be used.

Shortcoming #6 has been somewhat addressed via *regularization* (training penalties that discourage neuron *weights* from becoming very large), via *Dropout* layers (which temporarily and randomly eliminate some neurons at each iteration to force generalization), and other methods in recent years.

The remaining issues (1: sample efficiency, 2: data scarcity, 3: computational efficiency, 5: architecture search) still pose substantial challenges to the field. (Fancy GPU hardware is helping with #3.) More positively, they offer opportunities for the enterprising researcher!

Acknowledgements and thanks to Professors Mark Riedl and Jim Rehg of the Georgia Tech School of Interactive Computing.