It's all neural nets to me

Bob Carpenter, Flatiron Institute, carp@flatironinstitute.org

February 4, 2025

This note provides a statistician-friendly explanation of how deep neural networks provide a drop-in generalization of the linear predictor underlying a generalized linear model. To make the discussion concrete, we'll consider the industry-standard deep neural network built up by composing layers of perceptrons.

1 Perceptrons

A *perceptron* is a non-linear function $f : \mathbb{R}^M \to \mathbb{R}^N$ on real vectors defined by composing an affine transform h and a univariate non-linear "activation" function g applied elementwise. We will consider the "default" form of perceptron, which is

$$f(x) = \text{relu}(\alpha + \beta \cdot x),$$

with parameters $\alpha \in \mathbb{R}^N$ for the intercepts and $\beta \in \mathbb{R}^{N \times M}$ for the slopes. The *rectified linear unit* function $\text{relu} : \mathbb{R} \to \mathbb{R}$ is defined for $u \in \mathbb{R}$ by

$$\text{relu}(u) = \begin{cases} u & \text{if } u > 0, \text{ and} \\ 0 & \text{otherwise} \end{cases}$$

and extended to vectors by

$$\text{relu}(\begin{bmatrix} u_1 \cdots u_K \end{bmatrix}) = \begin{bmatrix} \text{relu}(u_1) \cdots \text{relu}(u_K) \end{bmatrix}.$$

There are also smooth activation functions with non-zero derivatives everywhere, such as the unbounded

$$\text{softplus}(u) = \log(1 + \exp(u)),$$

and the bounded

$$\text{logit}^{-1}(u) = 1/(1 + \exp(-u)).$$

2 Deep neural networks

A *deep neural network* (DNN) is the composition of a sequence of perceptrons,

$$dnn_{\alpha,\beta} = f_K \circ \cdots \circ f_1,$$

where the dimensions are conformal (i.e., the output dimensions of f_1 match the input dimensions of f_2 and so on). The *depth* of the neural network is K, and typically networks are deep with $K \gg 1$, typically $K \geq 10$ and sometimes much higher. The parameters α, β are now indexed by $1, \ldots, K$.

Sometimes we apply a so-called *layer norm* to control the location and scale of a layer. Typically this just standardizes the output vector to have zero mean and unit variance.

3 Generalized non-linear models

Adding a link function and sampling distribution to a linear model produces a *generalized linear model*. For example, if we have covariates $x_n \in \mathbb{R}^D$ and coefficients $\beta \in \mathbb{R}^D$, and observations $y_n \in \mathbb{N}$ consisting of counts, we can use a log link and Poisson likelihood, which looks as follows with the inverse of the link function.

$$y_n \sim \text{Poisson}(\exp(y_n \cdot \beta)).$$

3.1 Univariate generalized non-linear models

Suppose we have a deep neural network accepting N inputs and producing a single output, $\text{dnn}_{\alpha,\beta} : \mathbb{R}^D \to \mathbb{R}^1$. Such a network can be used for Bayesian unconstrained non-linear regression by assuming our sampling distribution is

$$y_n \sim \text{normal}(\text{dnn}_{\alpha,\beta}(x_n), \sigma).$$

If we have binary data, we can swap in the appropriate sampling distribution and link function to produce a logistic regression sampling distribution,

$$y_n \sim \text{bernoulli}(\text{logit}^{-1}(\text{dnn}_{\alpha,\beta}(x_n))).$$

Poisson and log link can be used for count data, etc.

3.2 Multivariate generalized non-linear models

Neural networks are popular as image classifiers. Images are represented as pixels in RGB space and these are converted to covariate vectors x_n. A 512×512 pixel image is rendered as a $(3 \times 512 \times 512)$-vector of intensity values.

A multi-layer neural network is then arranged to produce K outputs. These are run through softmax (a statistician might use the inverse isometric log ratio transform) to produce a simplex of probabilities which is the output expected from a probabilistic classifier,

$$y_n \sim \text{categorical}\big(\text{softmax}\left(\text{dnn}_{\alpha,\beta}(x_n)\right)\big)\,,$$

where $y_n \in \{1, \ldots K\}$, $x_n \in \mathbb{R}^D$, and $\beta \in \mathbb{R}^{K \times D}$, and $\text{softmax} : \mathbb{R}^N \to \Delta^{N-1}$ by

$$\text{softmax}(u) = \frac{\exp(u)}{\text{sum}(\exp(u))},$$

where $\Delta^{N-1} \subset \mathbb{R}^N$ is the set of $(N-1)$-dimensional unit simplexes (each of which has N components).

4 What about uncertainty?

From a high-level perspective, all we have done here is swap a liner function $x_n \cdot \beta$ for a non-linear function $\text{dnn}_{\alpha,\beta}(x_n)$. As such, it's business as usual for inference.

4.1 Penalized MLEs with confidence intervals

You can optionally put shrinkage priors on the slopes and intercepts α, β and calculate an MLE. This is often done implicitly in practice by stopping the neural network optimizer before it has converged.

Confidence intervals cannot be calculated analytically or even easily approximated, but they can be estimated through the bootstrap.

4.2 Bayesian inference

Other than compute cost, there's no obstacle to putting priors on the slopes and intercepts and sampling from Bayesian posteriors.[1]

5 Why deep neural networks?

Deep neural networks are attractive for two reasons: (1) in the infinite width limit, they are general function approximators, and (2) they are parallel compute friendly.

5.1 General function approximation

In the infinite-width limit, a neural network is a general function approximator. Consequently, they can be dropped in for just about any application for which there is enough data to estimate a general function. This is the same thing that motivates other black-box regression techniques like random forests, boosted decision trees, Bayesian additive regression trees (BART), and even Gaussian processes (GP).

5.2 Parallel compute friendly

The huge advantage neural nets have over competing black-box function approximators is that they are very parallel compute friendly. Moore's law, which is a conjecture that the density of transistors on chips will double every eighteen months, is still going strong. This isn't as evident as in the 1990s when our computers just got twice as fast every 18 months. Now every 18 months, they can do twice as many operations in parallel, not twice as many operations in a straight line.

This growth in computing is largely being realized on graphics processing units (GPUs). As I write this, a single state-of-the-art GPU, such as an NVIDIA H100 ($\approx$ US$50,000), can run over 50 teraflops (50 trillion floating point operations per second). That's as fast as the world's fastest computer 20 years ago and it's a single card that's expensive, but trivial to install. As an aside, the world's fastest computer today, El Capitan at Lawrence Liver-

[1] This is so popular that Matt Hoffman ran a large bake-off at NeurIPS after burning a ton of Google compute estimating a baseline.

more, runs at 2 exaflops, which is two quintillion floating point operations per second on over 1 million CPU cores and 10 million GPU cores.

There is an important caveat to using all these flops on a GPU. To exploit GPU parallelism, we have to use single-instruction multiple-data (SIMD) parallelism, which means executing the same operation on a lot of data in parallel. This repetitive structure is exactly what emerges from matrix arithmetic, for which GPUs are ideal.

6 Can it Stan?

It's just a regular old density, so of course it can be coded in Stan; see Figure 1 for the listing. We can even make it smooth by using softplus instead of relu activation. It's just not clear that Stan's sampling algorithms could sample this—the deep neural network community relies on stochastic gradient descent and a lot of data to fit these models.

Fitting with Stan is going to be tricky because the neural network is massively over-parameterized if $D \ll W$. Typically neural network models are fit with stochastic gradient descent optimization with early stopping.

In order to vectorize the loop over `N`, most neural network programming frameworks (e.g., PyTorch, JAX) allow operations to be batched across observations using tensor operations.

```
functions {
  vector relu(vector u) {
    return step(u) .* u;
  }
}
data {
  int<lower=0> N;
  int<lower=0> D;
  int<lower=0> W;
  matrix[N, D] x;
  array[N] int y;
}
parameters {
  vector[W] alpha1;
  matrix[W, D] beta1;

  vector[W] alpha2;
  matrix[W, W] beta2;

  real alpha3;
  row_vector[W] beta3;
}
model {
  for (n in 1:N) {
    // neural network by layer
    vector[D] input = x[n]';
    vector[W] layer1 = relu(alpha1 + beta1 * input);
    vector[W] layer2 = relu(alpha2 + beta2 * layer1);
    real output = alpha3 + beta3 * layer2;

    // likelihood, with inverse link
    y[n] ~ bernoulli(inv_logit(output));
  }

  // shrinkage priors
  alpha1 ~ std_normal();  beta1 ~ std_normal();
  alpha2 ~ std_normal();  beta2 ~ std_normal();
  alpha3 ~ std_normal();  beta3 ~ std_normal();
}
```

Figure 1: *Stan program implementing a neural net-logistic regression.*