AI1001 - Assignment 12

Shivram S
ai24btech11031@iith.ac.in

Deep learning is a family of machine learning techniques in which the hypotheses take the form of algebraic circuits with tunable connection strengths. Deep learning is currently the most widely used technique for applications such as visual object recognition, machine translation, speech recognition, etc. Networks trained by deep learning methods are often called **neural networks**. They are typically organized into **layers**.

1 Simple Feedforward Networks

A **feedforward network** has connections only in one direction. Each node computes a function of its inputs and passes the result to its successors. In neural networks, input values are typically continuous, and nodes take continuous inputs and produce continuous outputs. Some of the inputs are **parameters** of the network, and the network learns by adjusting these parameters.

Each node within a network is called a **unit**. Each unit calculates the weighted sum of its inputs, and applies a nonlinear **activation function** to produce its output.

$$a_j = g_j(\sum_i w_{i,j} a_i)$$

We can write this in vector form if we add an extra dummy input $a_0 = 1$, and a weight $w_{0,j}$ for that input.

$$a_j = g_j(\mathbf{w}^\top \mathbf{x})$$

A variety of different activation functions are used, such as

- The logistic/**sigmoid** function: $\sigma(x) = 1/(1 + e^{-x})$
- The **rectified linear unit** (ReLU): $\mathrm{ReLU}(x) = \max(0, x)$
- The **softplus** function: $\mathrm{softplus}(x) = \log(1 + e^x)$
- The **tanh** function: $\tanh(x) = (e^{2x} - 1)/(e^{2x} + 1)$

According to the **universal approximation** theorem, a network with just two layers can approximate any continuous function to an arbitrary degree of accuracy.

Combining multiple units together into a network creates a complex function that is a composition of algebraic expressions represented by individual units. A more general way to think about a network is as a **computation graph** or **dataflow graph**. For a two-layer network, the hypothesis can be represented as:

$$h_{\mathbf{w}}(\mathbf{x}) = \mathbf{g}^{(2)}(\mathbf{W}^{(2)}\mathbf{g}^{(1)}(\mathbf{W}^{(1)}\mathbf{x}))$$

Where $\mathbf{W}^{(n)}$ is the weight matrix and $\mathbf{g}^{(n)}$ is the activation function of the n^{th} layer, This expression corresponds to a computational graph which is **fully connected**. Choosing the connectivity of the network is also important in achieving effective learning.

Preprint. Under review.

2 Gradients and Learning

We can apply gradient descent to learning the weights in computational graphs. We can calculate the gradient for the network using the **chain rule**.

$$\frac{\partial g(f(x))}{\partial x} = g'(f(x))\frac{\partial f(x)}{\partial x}$$

For a weight, say $w_{3,5}$ connected to an output unit, we can calculate the gradient with respect to it as

$$\begin{aligned}
\frac{\partial}{\partial w_{3,5}} Loss(h_w) &= \frac{\partial}{\partial w_{3,5}}(y-\hat{y})^2 \\
&= -2(y-\hat{y})\frac{\partial}{\partial w_{3,5}} g_5(in_5) \\
&= -2(y-\hat{y})g_5'(in_5)\frac{\partial}{\partial w_{3,5}}(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4) \\
&= -2(y-\hat{y})g_5'(in_5)a_3
\end{aligned}$$

For a weight connected to a hidden layer, say $w_{1,3}$, we can calculate the gradient with respect to it as

$$\begin{aligned}
\frac{\partial}{\partial w_{1,3}} Loss(h_w) &= -2(y-\hat{y})g'(in_5)w_{3,5}\frac{\partial}{\partial w_{1,3}} g_3(in_3) \\
&= -2(y-\hat{y})g'(in_5)w_{3,5}g_3'(in_3)x_1
\end{aligned}$$

We can define $\Delta_5 = 2(\hat{y}-y)g_5'(in_5)$ as a sort of "perceived error" at the point where unit 5 represents its input, so that the gradient for $w_{3,5}$ as $\Delta_5 a_3$. Similarly, we can define $\Delta_3 = 2(\hat{y}-y)g_5'(in_5)w_{3,5}g_3'(in_3) = \Delta_5 w_{3,5}g_3'(in_3)$, so that the gradient for $w_{1,3}$ is $\Delta_3 a_1$. This phenomenon, where the error at the output is passed back through the network, is called **back-propagation**.

Gradient expressions have factors of the local derivatives $g_j'(in_j)$, which are always nonnegative, but can be very close to zero. As a result, deep networks with many layers may suffer from a **vanishing gradient**, where error signals are extinguished as they are propagated back.

Gradients can be calculated through **automatic differentiation**. For example, backpropagation uses **reverse-mode** differentiation, where the chain rule is applied from the outside-in. This has encouraged an approach called **end-to-end learning**, in which a complex computational system can be composed from several trainable subsystems. The entire system is trained in an end-to-end fashion from input-output pairs.

3 Computation Graphs for Deep Learning

The input and output nodes of a computational graph connect directly to the input data **x** and output $\hat{\mathbf{y}}$. The encoding of factored data is usually straightforward. Categorical attributes with more than two values are usually encoded using **one-hot encoding**. An attribute with d possible values is represented by d separate input bits. This ensures that all the possible values of the attribute are equidistant.

On the output side, ideally the prediction $\hat{y}$ would match the desired value **y**, and the loss would be zero. In practice, there is some error. It is common to interpret output values as probabilities and use **negative log likelihood** as the loss function. We look for the $\mathbf{w}^*$ that minimizes sum of negative log probabilities of the N examples:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} - \sum_{j=i}^{N} \log P_{\mathbf{w}}(\mathbf{y}_j \mid \mathbf{x}_j)$$

It is also common to talk about minimizing the **cross-entropy** loss, which is a measure of dissimilarity between two distributions P and Q.

$$H(P, Q) = \mathbf{E}_{\mathbf{z} \sim P(\mathbf{z})}[\log Q(\mathbf{z})] = \int P(\mathbf{z}) \log Q(\mathbf{z}) d\mathbf{z}$$

We typically use this definition with P being the true distribution over training examples $P^*(\mathbf{x}, \mathbf{y})$, and Q being the predictive hypothesis $P_{\mathbf{w}}(\mathbf{y}|\mathbf{x})$. Minimizing the cross-entropy $H(P^*(\mathbf{x}, \mathbf{y}), P_{\mathbf{w}}(\mathbf{y}|\mathbf{x}))$ by adjusting $\mathbf{w}$ makes the hypothesis agree with the true distribution. Even though we don't have the true distribution $P^*(\mathbf{x}, \mathbf{y})$, we have access to some samples from it, so we can approximate it to some degree.

In multiclass classification problems, we need the network to output a categorical distribution - if there are d possible answers, we need d output nodes that represent probabilities summing to 1. To achieve this, we use a **softmax** layer. The softmax function is smooth and differentiable, and the exponentials accentuate differences in the inputs.

$$\text{softmax}(\mathbf{in})_k = \frac{e^{in_k}}{\sum_{k'=1}^{d} e^{in_{k'}}}$$

Many other output layers are possible. For example, a regression problem might use a linear output layer $\hat{y}_j = in_j$ without any activation function g, and interpret this as the mean of a Gaussian prediction with fixed variance. A **mixture density** layer represents the output using a mixture of Gaussian distributions, and thus predicts the relative frequency of each mixture component.

4 Hidden layers

While processing an input vector $\mathbf{x}$, the neural network performs several intermediate computations before producing the output y. We can think of the values computed at each layer as a different *representation* for the input $\mathbf{x}$. Deep networks may form internal layers whose meaning is opaque to humans.

The hidden layers are typically less diverse than output layers. Internal nodes used sigmoid and tanh exclusively until around 2010, when ReLU and softplus became popular. Experimentation with increasingly deep networks suggest that better learning is obtained with deep and relatively narrow networks.