Not Just A Black Box:
Interpretable Deep Learning by Propagating Activation Differences

Abstract

A common criticism of neural networks is their lack of interpretability or "black box" nature. This is a major barrier to adoption in application domains such as biology, where interpretability is important. Here we present DeepPAD (Propagating Activation Differences), an intuitive and effective method for scoring the contributions of inputs to the output of a neural network. DeepPAD compares the activation of each neuron to its 'reference activation', and assigns a contribution score to each of the neuron's inputs proportionally to how the inputs differ from their own 'reference activations'. We apply DeepPAD to models trained on natural images as well as models trained on genomic sequences. We show that correctly identifies important features in both cases and has significant advantages over gradient-based methods.

1. Introduction

As neural networks become increasingly popular, their reputation as a "black box" presents a barrier to adoption in fields were interpretability is paramount. Understanding the features that lead to a particular output builds trust with users and can lead to novel scientific discoveries. A common approach is to leverage the gradients of a particular output with respect to the individual inputs (Simonyan et al. - cite) - however, such approaches are limited because activation functions such as Rectified Linear Units (ReLUs) have a gradient of zero when they are not firing, yet a ReLU that does not fire can still carry information - particularly if the associated bias is positive (which means that in the absence of any input, the ReLU does fire). Similarly, sigmoid or tanh activations are popular choices for the activation functions of gates in memory units of recurrent neural networks such as GRUs and LSTMs (cite), but these activations have a near-zero gradient at high or low inputs even though such inputs can be very significant.

In this paper, we present a general method for assigning feature importance which focuses on the difference between a neuron's activation and its 'reference' activation, where the reference activation is the activation that the neuron has when the network is provided a 'reference input'. The reference input is defined on a domain-specific basis according to what is appropriate for the task at hand. This circumvents the aforementioned limitation of gradient-based approaches because, rather than relying on the local gradient at the point of activation, the activation is simply compared to its reference value. In the case of a ReLU that fires when provided the reference input (as is typically the case for ReLUs that have a positive bias), our method assigns a negative 'difference from reference' when the ReLU does not fire. Similarly, in the case of a sigmoidal unit which has an output of 0.5 when provided the reference input, our method would assign a positive 'difference from reference' when the sigmoidal unit has an output near 1, even though the gradient at that output is negligible.

2. DeepPAD Method

We denote the contribution of x to y as C_{xy}. Let the activation of a neuron n be denoted as A_n. Further, let the *reference* activation of neuron n be denoted A_n^0, and let the $A_n - A_n^0$ be denoted as δ_n. We define our contributions C_{xy} to satisfy the following properties.

2.1. Summation to δ

For any set of neurons S whose activations are minimally sufficient to compute the activation of y (that is, if we know the activations of S, we can compute the activation of y, and there is no set $S' \subset S$ such that S' is sufficient to compute the activation of y - in layman's terms, S is a full set of non-redundant inputs to y), the following property holds:

$$\sum_{s \in S} C_{sy} = \delta_y \tag{1}$$

That is, the sum over all the contributions of neurons in S to y equals the difference-from-reference of y.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

2.2. Linear composition

Let O_x represent the output neurons of x. The following property holds:

$$C_{xy} = \sum_{o \in O_x} \frac{C_{xo}}{\delta_o} C_{oy} \tag{2}$$

In layman's terms, each neuron 'inherits' a contribution through its outputs in proportion to how much that neuron contributes to the difference-from-reference of the output.

2.3. Backpropagation Rules

We show that the contributions as defined above can be computed using the following rules (which can be implemented to run on a GPU). The computation is reminiscent of the chain rule used during gradient backpropagation, as equation 2 makes it possible to start with contribution scores of later layers and use them to find the contribution scores of preceding layers. To avoid issues of numerical stability when δ_n for a particular neuron is small, rather than computing the contribution scores explicitly, we instead compute *multipliers* m_{xy} that, when multiplied with the difference-from-reference, give the contribution:

$$m_{xy}\delta_x = C_{xy} \tag{3}$$

Let t represent the target neuron that we intend to compute contributions to, and let O_x represent the set of outputs of x. We show that:

$$m_{xt} = \sum_{y \in O_x} m_{xy} m_{yt} \tag{4}$$

The equation above follows from the linear composition property and the definition of the multipliers, as proved below:

$$\begin{aligned}
C_{xt} &= \sum_{y \in O_x} \frac{C_{xy}}{\delta_y} C_{yt} \\
m_{xt}\delta_x &= \sum_{y \in O_x} \frac{C_{xy}}{\delta_y} (m_{yt}\delta_y) \\
&= \sum_{y \in O_x} C_{xy} m_{yt} \\
m_{xt} &= \sum_{y \in O_x} \frac{C_{xy}}{\delta_x} m_{yt} \\
m_{xt} &= \sum_{y \in O_x} m_{xy} m_{yt}
\end{aligned} \tag{5}$$

In the equations below, I_y denotes the set of inputs of y.

2.3.1. Affine functions

Let

$$A_y = \left(\sum_{x \in I_y} w_{xy} A_x \right) + b \tag{6}$$

Then

$$m_{xy} = w_{xy} \tag{7}$$

Proof. We show that

$$\delta_y = \sum_{x \in I_y} m_{xy} \delta_x \tag{8}$$

Using the fact that $A_n = A_n^0 + \delta_n$, we have:

$$\begin{aligned}
(A_y^0 + \delta_y) &= \left(\sum_{x \in I_y} w_{xy}(A_x^0 + \delta_x) \right) + b \\
&= \left(\sum_{x \in I_y} w_{xy} A_x^0 \right) + b + \sum_{x \in I_y} w_{xy}\delta_x
\end{aligned} \tag{9}$$

We also note that the reference activation A_y^0 can be found as follows:

$$A_y^0 = \left(\sum_{x \in I_y} w_{xy} A_x^0 \right) + b \tag{10}$$

Thus, canceling out A_y^0 yields:

$$\begin{aligned}
\delta_y &= \sum_{x \in I_y} w_{xy}\delta_x \\
&= \sum_{x \in I_y} m_{xy}\delta_x
\end{aligned} \tag{11}$$

2.3.2. Max pooling over convolutional filters

We consider the case of maxpooling over a set of neurons in a convolutional layer which all belong to the same convolutional filter:

$$A_y = \max_{x \in I_y} A_x \tag{12}$$

We make the assumption that all instances of a single convolutional filter have the same reference activation, i.e. A_x^0 is the same for all x in I_y (this would not hold if the chosen reference input were not uniform across all positions; the appropriate multiplier in that situation can be computed easily, but is omitted here for brevity). Then we have:

$$m_{xy} = \mathbf{1}\{A_x = A_y\} \tag{13}$$

Where $\mathbf{1}\{\}$ is the indicator function. If a symbolic computation package is used, then the gradient of y with respect to x can be used in place of $\mathbf{1}\{A_x = A_y\}$.

Proof. Because A_x^0 is assumed to be the same for all x, we have that $A_y^0 = A_x^0$. As $A_y = A_y^0 + \delta_y$, we get:

$$\begin{aligned}
\delta_y &= A_y - A_y^0 \\
&= A_y - A_x^0 \\
&= \mathbf{1}\{A_x = A_y\}(A_x - A_x^0) \\
&= \mathbf{1}\{A_x = A_y\}\delta_x \\
&= m_{xy}\delta_x
\end{aligned} \tag{14}$$

2.3.3. Individual nonlinearity

Here we describe the rule for nonlinear transformations such as the ReLU, sigmoid and tanh which are applied to individual inputs, typically following some affine transformation. We have:

$$A_y = f(A_x) \tag{15}$$

Where f is the nonlinear transformation. When δ_x is large in magnitude, we can compute m_{xy} according to its definition as follows:

$$m_{xy} = \frac{\delta_y}{\delta_x} \tag{16}$$

When δ_x is small, the term on the right approaches $\frac{0}{0}$. However, as $\delta_y \to 0$ as $\delta_x \to 0$, we have:

$$\lim_{\delta_x \to 0} \frac{\delta_y}{\delta_x} = \frac{d\delta_y}{d\delta_x} \tag{17}$$

Thus, when δ_x is close to zero, m_{xy} is simply the derivative of δ_y with respect to δ_x. Because $\delta_n = A_n - A_n^0$ and A_n^0 is a constant, this is the same as $\frac{dA_y}{dA_x}$ when δ_x is near 0.

2.3.4. Softmax activation

Let $t_1, t_2 ... t_n$ represent the output of a softmax transformation on the nodes $y_1, y_2 ... y_n$, such that:

$$A_{t_i} = \frac{e^{A_{y_i}}}{\sum_{i'=1}^{n} e^{A_{y'_i}}} \tag{18}$$

Here, $A_{y_1} ... A_{y_n}$ are affine functions of their inputs. Let x represent a neuron that is an input to $A_{y_1} ... A_{y_n}$, and let w_{xy_i} represent the coefficient of A_x in A_{y_i}. Because $A_{y_1} ... A_{y_n}$ are followed by a softmax transformation, if w_{xy_i} is the same for all y_i (that is, x contributes equally to all y_i), then x effectively has zero contribution to A_{t_i}. This can be observed by substituting $A_{y_i} = w_{xy_i} A_x + r_{y_i}$ in the expression for A_{t_i} and canceling out $e^{w_{xy_i} A_x}$ (here, r_{y_i} is the sum of all the remaining terms in the affine expression for A_{y_i})

$$\begin{aligned} A_{t_i} &= \frac{e^{A_{y_i}}}{\sum_{i'=1}^{n} e^{A_{y'_i}}} \\ &= \frac{e^{w_{xy_i} A_x + r_{y_i}}}{\sum_{i'=1}^{n} e^{w_{xy'_i} A_x + r_{y'_i}}} \\ &= \frac{e^{w_{xy_i} A_x + r_{y_i}}}{\sum_{i'=1}^{n} e^{w_{xy_i} A_x + r_{y'_i}}} \\ &= \frac{e^{r_{y_i}}}{\sum_{i'=1}^{n} e^{r_{y'_i}}} \end{aligned} \tag{19}$$

One way to ensure that C_{xt} is zero if w_{xy_i} is the same for all y_i is to set:

2.4. A note on final activation layers

2.5. A note on weight normalization for constrained inputs

3. Results

4. Discussion

5. References