Proving robustness with DeepZ relaxation

Abhinav Aggarwal & Meet Vora

December 20, 2019

Abstract

Based on the idea of Abstract Interpretation, the authors of DeepZ propose new abstract transformers, which use the Zonotope abstraction. In our work, we employ the abstract transformers for linear transformations and ReLU activations. We implement their work to verify simple feed-forward networks and convolutional neural networks, both of which are trained on MNIST.

Formulation

From the paper, we know that with Zonotope relaxation, we can represent any given activation as

$$\tilde{x}^j := \alpha_0^j + \sum_{i=1}^{k} \alpha_i^j \epsilon_i$$

where ϵ vary within $[-1, 1]$ and are shared between activations/neurons. Based on this, when we perform any linear transformation (i.e. $y = Wx + b$), we obtain $\tilde{y} = (W\alpha_0 + b) + W \times (\sum_{i=1}^{k} \alpha_i \epsilon_i)$.

Implementation

We treat a given input x as a point in a (k+1)-dimensional vector space where each constant(α_i) denotes the magnitude along that ϵ_i axis, i.e. $x = (\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_k)$. To do so, firstly, we transform the input condition (image and epsilon) to a box transformation. Thus, we have 784 epsilon terms to start with, of the form: $\tilde{x}_i = x_i + \epsilon_i$ (one epsilon for each input neuron). Presenting this in vector form we get: $\tilde{x}_i = (x_i, 0, 0, \ldots, \epsilon_i, 0, 0, \ldots)$ where our $\tilde{x}_i$ is now a 785-dimensional vector. (Note that x_i and ϵ_i are modified such that their $x_i + \epsilon_i$ doesn't exceed 1 and $x_i - \epsilon_i$ doesn't go subzero.)

We know that `PyTorch` requires the inputs to any `nn.Module` to be of the shape (N, C, H, W). We use this fact to transform our original input (a 784-dimensional vector) to an input tensor of shape similar to (784, 785). For each epsilon, we create a 784-dimensional vector (and thus a total of 784 such vectors). Let's discuss an example to clarify our point. Since only $\tilde{x}_1$ contains ϵ_1 and the contribution of ϵ_1 in all other x_i is 0, we can create a 784-dimensional zero-vector which is set only at its first index. And repeat the same for each i. We concatenate all of these vectors to finally create a tensor of shape similar to (784, 785) where each row represents $\tilde{x}_i$. Practically, we do this concatenation along the batch-axis and thus, we obtain our input in the shape (785, 1, 28, 28). We can thus interpret our final input as a batch of images instead of a single image.

Extending the formulation of linear transformation above, we expect to obtain $y = (W\alpha_0 + b, W\alpha_1, W\alpha_2, \ldots, W\alpha_k)$ with a linear transformation of form $y = Wx + b$. To obtain the same result with our input, we create our own modified linear layer that replaces the original linear layer, such that the bias term gets added only to the first image in the batch. The affine transformation takes place uniformly across all images. Similarly, we create a modified convolutional layer which does the same affine transformation for all images of the batch, but adds bias only to the first image. This idea works because a convolutional layer works like a locally-connected linear layer.

For ReLU, we again create a modified layer which performs the Zonotope ReLU transformation. Each ReLU layer has a *slope*, *intercept*, *upper_bound* and *lower_bound*. Each of these is a vector of same length as the number of neurons in the given layer. Thus $slope_i$ represents the slope of i^{th} neuron in our layer. Note that while *upper_bound* and *lower_bound* are functions of the inputs, slope and intercept are the parameters of our system which we wish to optimize such that our model is correctly verified at most points. During the first forward-pass through our new model (with modified layers), we initialize each ReLU neuron with the minimum area slope configuration as discussed in the paper, in case a crossing takes place. We know that a new epsilon is added to the system if a crossing takes place. Thus, for each such neuron, we append another 784-dimensional vector to our input (along the batch-axis), populated with values obtained from the formula discussed in the paper. Note that *intercept* is a function of *slope*, *lower_bound* and *upper_bound*. If slope $(\lambda) \in [\frac{u_x}{u_x - l_x}, 1]$, we have $intercept = -\lambda * l_x$, else $u_x(1-\lambda)$. To allow these updates dynamically during runtime, we implement intercept as a read-only property for our ReLU layer.

Verification

Using each ϵ_i as a different image in the batch allows us to obtain a final tensor (z) of the shape $(num\_final\_epsilons, 10)$. Thus, to obtain the final activation for class i, we simply need to read $z[:, i]$, which returns a $num\_final\_epsilons$-dimensional vector. To check if our true label j is always the most activated, we can simply check the minimum possible value of scalar represented by $(z[:, j] - z[:, i])$ for all i. If we obtain non-negative values for all classes, we claim that the model is verified at our input.

Optimization

In order to optimize slope and intercept, we create a loss function that is inspired by hinge-loss. The loss is of the form

$$\sum_{i=1}^{10} \max(-\delta_i, 0)$$

where δ_i is the minimum possible value of scalar represented by $(z[:, j] - z[:, i])$ and j is our true label. The loss is structured such, so as to ignore the δ obtained from neurons which are already less activated than our true label. We minimize this loss with Adam optimizer, which optimizes only the *slope* and *intercept* of all ReLU layers. The training stops when loss reduces to 0, giving us a configuration that verifies our model for the input. Note that we ensure that the slope is always between 0 and 1.