Army Research Lab: Neural networks for physical dynamics

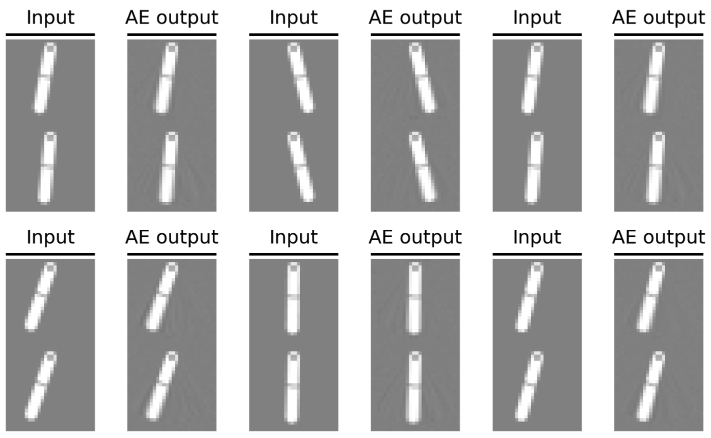
Papers read:

"Exact solutions to the nonlinear dynamics of learning in deep linear neural networks" (https://arxiv.org/pdf/1312.6120.pdf)

Given that the autoencoder in the Hamiltonian Neural Networks paper uses a particular initialization technique that requires orthogonalized weights, we wanted to know why this might be the case. PyTorch developed this initialization method: `torch.nn.init.orthogonal_()` in accordance with the paper "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks". In this paper, the authors show that random scaled Gaussian initializations can not achieve dynamical isometry despite their norm-preserving nature, while random orthogonal initialization can, thereby achieving depth independent learning times.

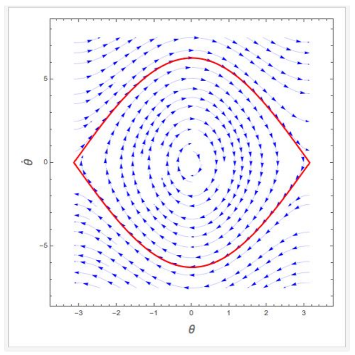
Customized Weight Initialization

We wanted to compare the autoencoder with and without the orthogonal initialization to see how much of an impact it would have on the autoencoder's performance. We trained the autoencoder with the same parameters only getting rid of the initialization and these were the results:

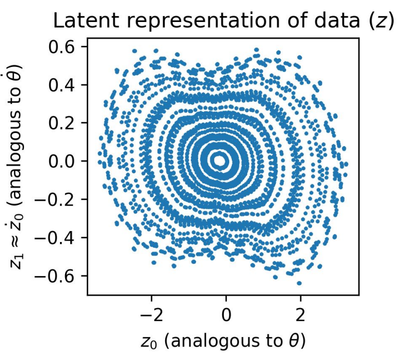


The output does not look too bad, but after examining the phase space representation, we can see that the autoencoder's performance has worsened.

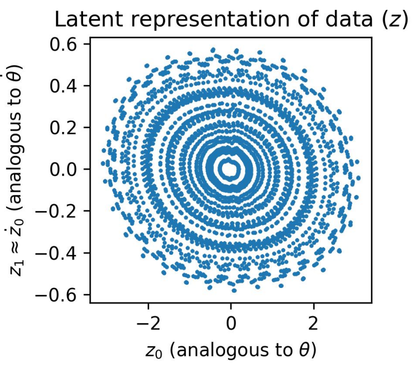
A simple pendulum's phase space should look like this:



The AE without weight init

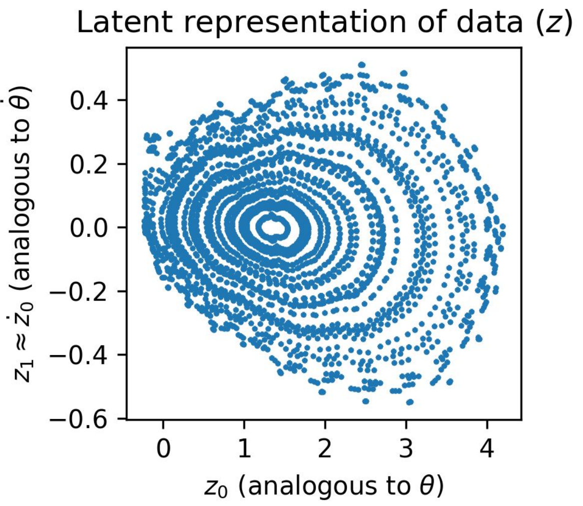


The AE with with weight init



Residual Connections

We also wanted to test the importance of residual connections, by also comparing the autoencoder's performance with and without residual connections. We trained the autoencoder without adding up the previous layer's activations. These were the results:



Runge-Kutta Method to Solve ODEs

The Runge-Kutta methods are iterative methods to solve ODEs. Given an ODE that defines a value of dy/dx in the form of x and y and an initial value of u, i.e. y(0), we are given:

$$\frac{dy}{dt} = f(t,y),\ y(0) = y_0 .$$

The task for this problem is to find the value of the unknown function y (scalar or vector) at any given time t. We are told that dy/dt, the rate at which y changes, is a function of t and of y. At the initial time t_0 the corresponding y value is y_0. The Runge-Kutta method finds the approximate value of y for a given t.

The formula used to compute the next value y_{n+1} from the previous value y_n, where n = 1,2,3, ... Here, h is the step size and positive (> 0). Lower step size means more accuracy.

$$y_{n+1} = y_n + \frac{1}{6}h\left(k_1 + 2k_2 + 2k_3 + k_4\right),$$
$$t_{n+1} = t_n + h$$

n = 1, 2, 3, … using,

$$\begin{aligned}
k_1 &= f(t_n, y_n), \\
k_2 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}\right), \\
k_3 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}\right), \\
k_4 &= f\left(t_n + h, y_n + hk_3\right).
\end{aligned}$$

The formula essentially computes the next value y_{n+1} using the current value y_n plus weighted average of four increments.

- k_1 is the increment based on the slope at the beginning of the interval, using y (Euler's Method).
- k_2 is the increment based on the slope at the midpoint of the interval, using y and k_1.
- k_3 is again the increment based on the slope at the midpoint, using y and k_2.
- k_4 is the increment based on the slope at the end of the interval, using y and k_3.

The method is a fourth-order method, meaning that the local truncation error is on the order of $O(h^5)$, while the total accumulated error is order $O(h^4)$.

Residual Neural Networks Justification in image recognition problems:

See the following paper: Deep Residual Learning for Image Recognition

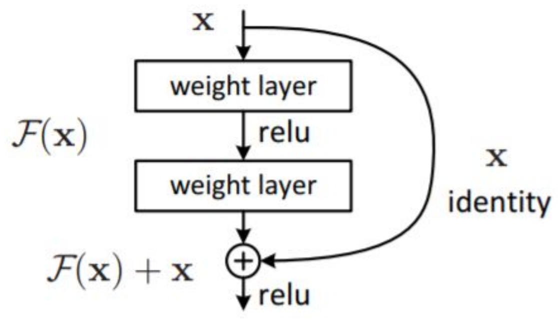
Summary:

Deep neural networks have been integral to the success of large and challenging datasets, such as COCO or ImageNet datasets. A great example of this is BERT in NLP. However, simply increasing the depth of a neural network doesn't necessarily come with improvements in the test/validation error; in some cases, increasing the depth of a neural network increases the validation error in a way that can't be explained by overfitting. One problem is that (depending on the nonlinear transformation of the layers) as the depth of the network increases, the gradients for the first layers either vanish or explode. This has largely been solved by batch normalization layers. However, the trouble still remains -- that even without vanishing/exploding gradients of earlier layers, some deeper neural networks are harder to train than shallower ones (by virtue of the following figure).

It's easy to construct a solution to this problem; If a shallower network performs better, we can construct a deep neural network with the same performance by just making the first layers identity mappings, and placing the shallow network on top. The existence of such a construction implies that, in principle, deep neural networks shouldn't perform any worse than a shallow network. This does not appear to be the case (again, see the figure above). Residual neural networks attempt to solve this problem, by allowing the network (in some sense) to learn how "deep" of a network is required.

Fundamentally, neural networks (F_w(x)) are trying to learn some underlying remapping of the data (H(x)). The author's of the paper posit that learning the residual map (optimize w in F_w(x) = H(x)-x) is easier to learn than the original mapping (optimize w in F_w(x) = H(x)). (an important note here is that for very easy implementation, you need to make sure that the input dimensions match the output dimensions to apply this technique. This happens to be perfectly fine for the video encoder of our HNN, being a collection of stacked densely connected 28*28 layers). The author's example is learning the identity mapping (H(x) = x) is harder to do through a collection of nonlinear functions, than learning the residual (H(x) - x = 0). All a residual learner has to do is drive the network weights for the non-linear component to 0.

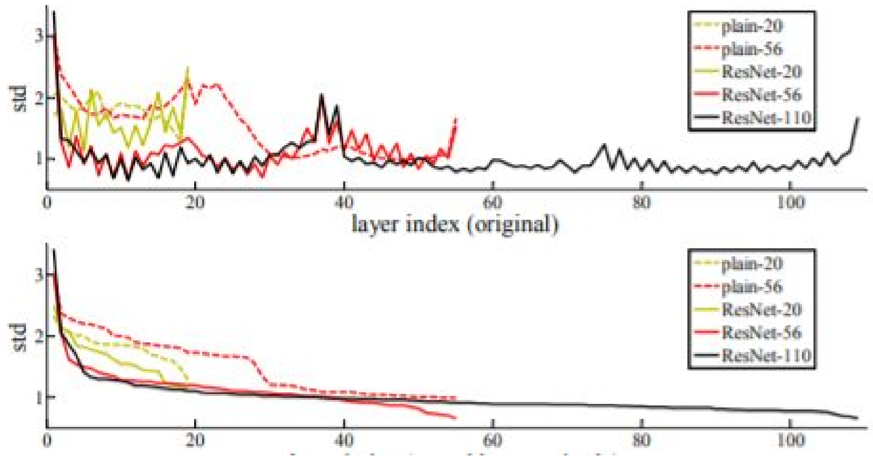
There's good evidence that in vector quantization, it's easier (and more compact) to learn the residual vectors; moreover, it's been shown that numerical PDE solvers that incorporate residuals converge faster than the PDE solvers which don't. In some sense, newtonian methods (which incorporate the hessian) use residual information for how to update parameters, and converge faster (w.r.t. total number of updates) than gradient descent or SGD algorithms.


x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

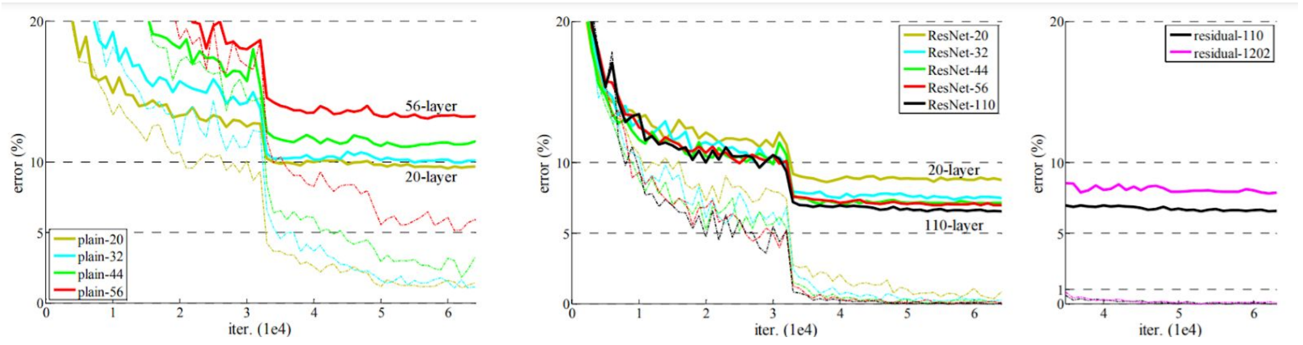
From a practical standpoint, learning the residuals is generally done through sub-blocks shown to the left. A notable advantage, is that there is minimal added computational cost for computing a forward pass, gradients, and weight updates

Moreover, it's possible for the inputs/output sizes to not match, as we can approximate the residuals via a linear transformation (H(x) - Mx, where M is a matrix of size Dim(H(x))by Dim(x)). However, the weights of this must be learned, or must be done through a projection.

The following chart further suggests that residual networks tend to have smaller modifications of the inputs (i.e., once trained the weights are closer to 0), which potentially suggests that residual networks only need to explore a smaller radius in parameter space -- this is further confirmed by the somewhat faster convergence of similar sized Residual networks against plain networks.



Finally, there’s still a limitation for how deep you can make a residual neural network before you start to get performance loss again (which can’t be explained by overfitting). The next graphic compares plain neural networks and residual neural networks on an image dataset for various depths. Increasing the depth to over 1200 layers resulted in such unexplainable losses (see the rightmost figure).


plain-20
plain-32
plain-44
plain-56
56-layer
20-layer
ResNet-20
ResNet-32
ResNet-44
ResNet-56
ResNet-110
110-layer
residual-110
residual-1202
error (%)
iter. (1e4)