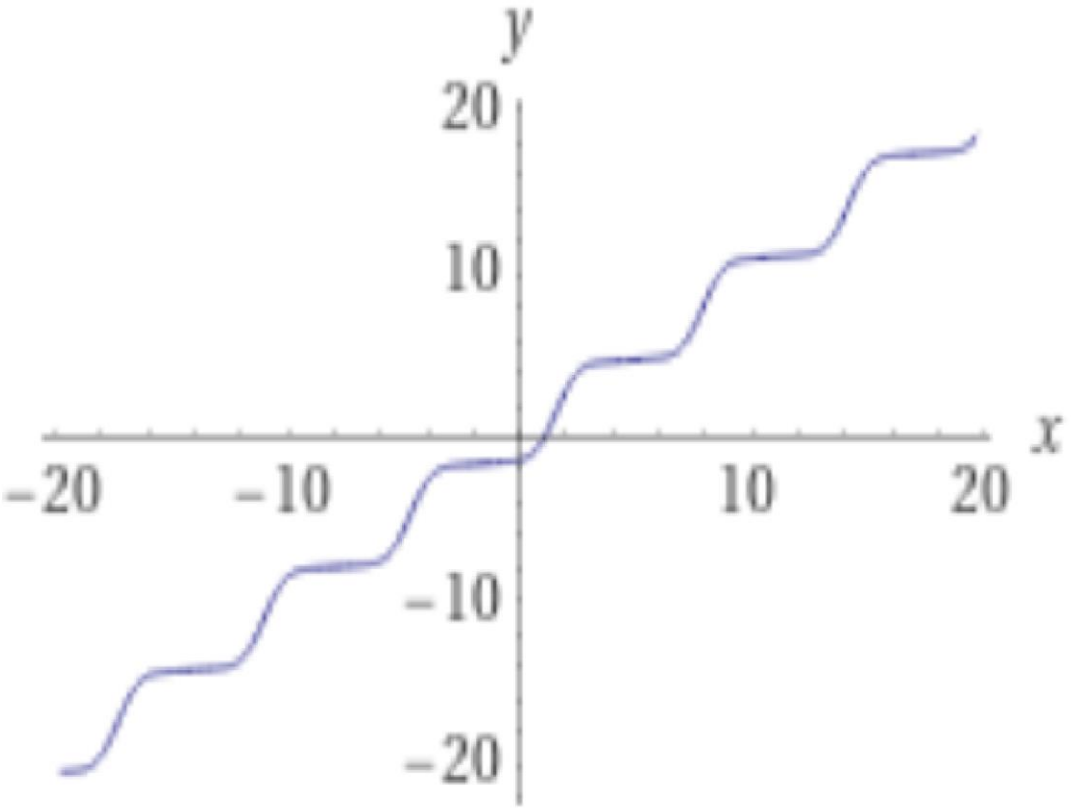
Novel Activation Function

As a self-identified mathematician data scientist, I have always thrived when I can rigorously prove that an algorithm will behave according to a well-defined framework under a given set of assumptions, and as awesome as a rigorously proven algorithm is, sometimes we need do what I call 'proof by hand waving'. In this article, I will attempt to introduce a novel activation function, and while its veracity is not rigorously proven, I will present empirical evidence that this new activation function seems to work well for certain problems.

To begin, let's talk about activation functions in general. The most basic neural network is a dense neural network (DNN), and it is created by stringing multiple matrix multiplications together until we get our final output; however, if we only do matrix multiplications, it can easily be shown that the output is equivalent to a single matrix multiplication (this is because matrix multiplication is associative, so we can just do all the middle multiplications first, and that results in a brand new matrix that we can use for our single matrix multiplication). To overcome this issue, we can introduce an activation function after each multiplication. The only requirement of an activation function is that it cannot be linear. In one dimension, this is equivalent to saying our activation function cannot be a straight line. There have been many activation functions proposed, and only a few will be mentioned here. First, we have the step function. A step function takes on a single value for a given interval, and then immediately jumps to a new value once the interval is left. Two textbook examples of a step function are the floor and ceiling functions (round down/up to the nearest integer). This activation function would be nice in theory because it allows inputs to be very stable for a given interval before jumping to a new value; however, this activation function should never be used in practice because its derivative is zero almost everywhere, and in the countable instances in which it is not zero, the derivative is undefined (good luck doing gradient descent on that function) Another activation function is the sigmoid function. This function is given by $S(x) = \frac{1}{1+e^{-x}}$. This function can be compared to a continuous everywhere differentiable version of a step function where it only has one step from zero to one. A big drawback of this function is that its derivative quickly approaches zero as the input get farther from zero; so while the derivative may be non-zero, it can be so close to zero that it takes a really long time to find an optimal solution via gradient descent. The final popular activation function I will touch on here is rectified linear units (RELU). This function has been shown empirically to be a very powerful activation function, and it is equal to 0 for all input less than 0, and is the identity function otherwise. RELU seems very odd as a good choice because its derivative is either 0 (a bad thing as mentioned), or the function is completely linear (also a bad thing), and yet somehow, combining two bad things results in a good thing? The evidence clearly shows that RELU does work very well for a wide variety of tasks, but I still often wonder if there is a better choice out there, and that is the goal of this investigative article.

In considering a new novel activation function, I am drawn to the concept of the step function. We cannot use the step function for reasons already mentioned, but could we get something that is everywhere differentiable, and approximates a step function? One candidate that immediately comes to mind is $\sin(x) + x$, but this choice has a couple problems. I'd optimally like to see a function that quickly rises from one level to another, the proposed function here has a derivative that has a maximum of 2. We can increase this maximum by multiplying $\sin(x)$ by a value greater than 1, but this results in an infinite number of critical points which makes finding an optimal value difficult, we can also multiply the linear term by a value greater than one, but this makes the function more closely approximate a linear function, which we do not want. Is there a way to create a function whose derivative is always greater than 0 (this ensures we do not have any critical points), is near zero for a set interval (This makes the function generally stable in that interval), and then quickly rises to a new high level before quickly returning to near zero (This makes the function approximate the step function)? In addition, it would be nice to have the derivative average 1 (that way we do not blow up our inputs and we're centered around the identity function). At this point, we are describing the derivative more than we are describing the actual function, so perhaps we can find the derivative, and then integrate the derivative to get the actual function we are interested in. There are many different functions that meet these requirements, and I would love to hear from readers about alternatives they may find, but one function I have found follows the form $((\sin(x) + \alpha)^4 + \mu) * \beta$ where alpha is greater than 1 (This makes the derivative always greater than 0), mu is greater than 0 (this is useful if we want to make the derivative more than trivially greater than 0), and beta is just a factor to makes the average equal to 1. For my example, I have set alpha to 1.1, mu is 0.6 and beta is approximately 0.165. The integral of this function is well defined - and while I'd love to say I calculated it analytically myself - I ended up using wolfram alpha. The integral (remember our activation function is the integral) evaluates to be approximately $1.0015x - .340759\sin(2x) + .00515677\sin(4x) - 1.4321\cos(x) + .0605061\cos(3x)$. Below is what the function looks like in one dimension.



Now that we have our proposed new activation function with all the desired properties, let's start exploring it a bit more. Before I really diving into getting empirical evidence, I would like to implement a neural network from scratch. I like to do this when exploring new algorithms because it forces me to really understand the nuts and bolts of what is going on. The interested reader can see the code on my GitHub (linked at the end of this article), but TLDR is that I implement a working example using numpy, and stochastic gradient descent seems to be converging to an optimal point, but the loss is much higher in my model from scratch compared to my pytorch model, likely due to the fact that I only included 10k training samples (due to slower compute time in my scratch model).

The next step is to fully train a model using state of the art tools (I will be using pytorch, but other perfectly reasonable options exist such as tensorflow). I start by defining the neural net class for this algorithm

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.L1 = nn.Linear(10,880,bias=False)
        self.L2 = nn.Linear(880,1,bias=False)

    def forward(self, x):
        l1 = self.L1(x)
        l2 = l1-1.4231*torch.cos(l1)+0.0605061*torch.cos(3*l1)-0.340759*torch.sin(2*l1)+0.00515677*torch.sin(4*l1)
        return self.L2(l2)
```

I then instantiate the model and define the loss function (mean squared error) and optimizer (stochastic gradient descent).

```
model = Net()
```

```
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.1)
```

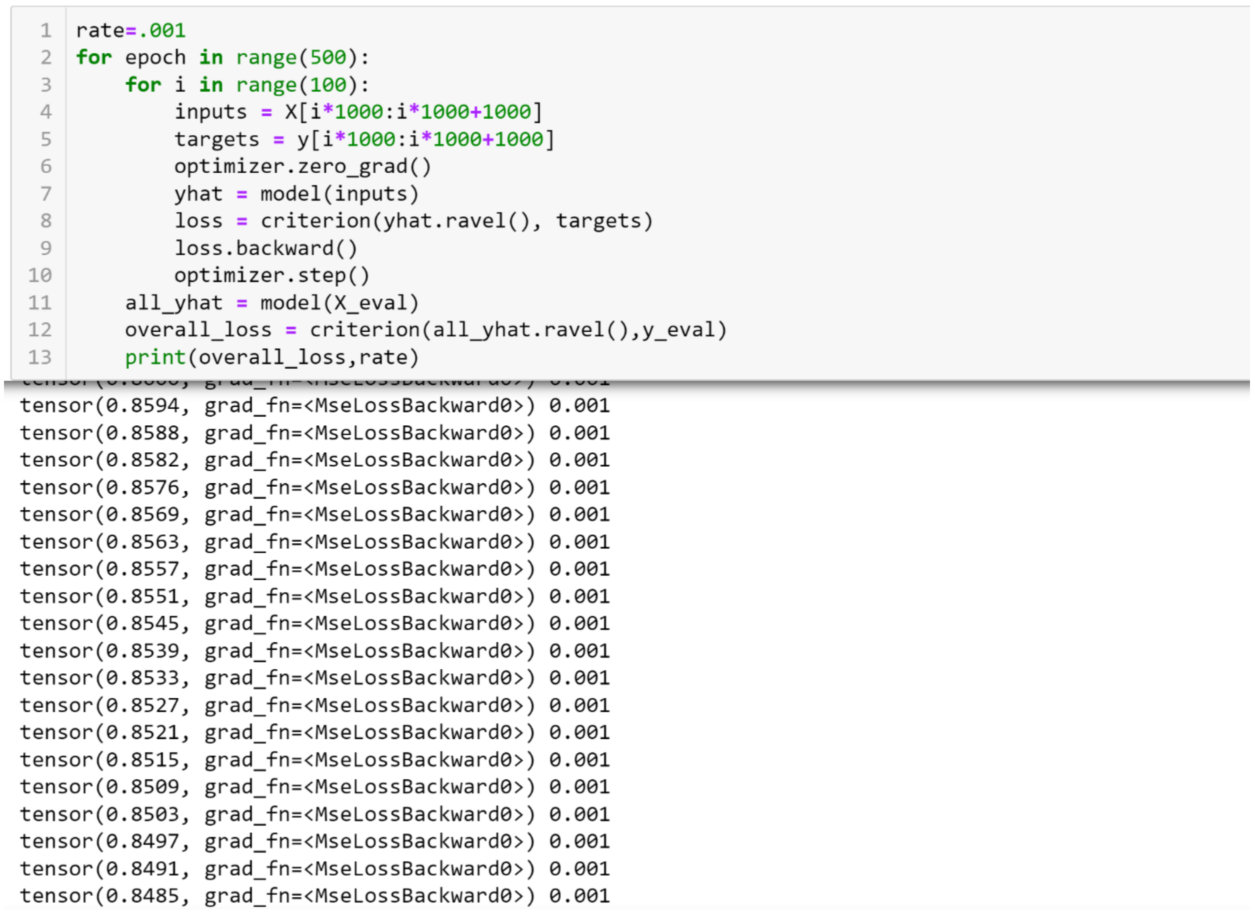
I also need some data to train on. When exploring new algorithms, I will often create synthetic test data because then I know exactly how the targets are related to the input features, and I can create as many samples as I want. The exact relationship isn't too important, but we are able to verify that some sort of relationship does exist.

```
X = np.random.randn(100000,10)
y=(np.sin(X)/np.exp(X)).sum(axis=1)
X=torch.tensor(X).float()
y=torch.tensor(y).float()
```

```
X_eval = np.random.randn(1000,10)
y_eval=(np.sin(X_eval)/np.exp(X_eval)).sum(axis=1)
X_eval=torch.tensor(X_eval).float()
y_eval=torch.tensor(y_eval).float()
```

In the example I've given above, the relationship is exact (no random term), but since I'm dividing by something that can potentially be near zero, the relationship will be unstable, so it should give a relationship that is moderately difficult to model.

Now we are ready to train our model with our new activation function.



```
rate=.001
for epoch in range(500):
    for i in range(100):
        inputs = X[i*1000:i*1000+1000]
        targets = y[i*1000:i*1000+1000]
        optimizer.zero_grad()
        yhat = model(inputs)
        loss = criterion(yhat.ravel(), targets)
        loss.backward()
        optimizer.step()
    all_yhat = model(X_eval)
    overall_loss = criterion(all_yhat.ravel(),y_eval)
    print(overall_loss,rate)
```

```
tensor(0.8594, grad_fn=<MseLossBackward0>) 0.001
tensor(0.8588, grad_fn=<MseLossBackward0>) 0.001
tensor(0.8582, grad_fn=<MseLossBackward0>) 0.001
tensor(0.8576, grad_fn=<MseLossBackward0>) 0.001
tensor(0.8569, grad_fn=<MseLossBackward0>) 0.001
tensor(0.8563, grad_fn=<MseLossBackward0>) 0.001
tensor(0.8557, grad_fn=<MseLossBackward0>) 0.001
tensor(0.8551, grad_fn=<MseLossBackward0>) 0.001
tensor(0.8545, grad_fn=<MseLossBackward0>) 0.001
tensor(0.8539, grad_fn=<MseLossBackward0>) 0.001
tensor(0.8533, grad_fn=<MseLossBackward0>) 0.001
tensor(0.8527, grad_fn=<MseLossBackward0>) 0.001
tensor(0.8521, grad_fn=<MseLossBackward0>) 0.001
tensor(0.8515, grad_fn=<MseLossBackward0>) 0.001
tensor(0.8509, grad_fn=<MseLossBackward0>) 0.001
tensor(0.8503, grad_fn=<MseLossBackward0>) 0.001
tensor(0.8497, grad_fn=<MseLossBackward0>) 0.001
tensor(0.8491, grad_fn=<MseLossBackward0>) 0.001
tensor(0.8485, grad_fn=<MseLossBackward0>) 0.001
```

The model is certainly able to generalize to data not included in the training set as evidenced by the dropping MSE score, but the real question is how does this stack up against the established players in machine learning? To answer this question, I created a new model with the exact same architecture but replacing the activation function with RELU.

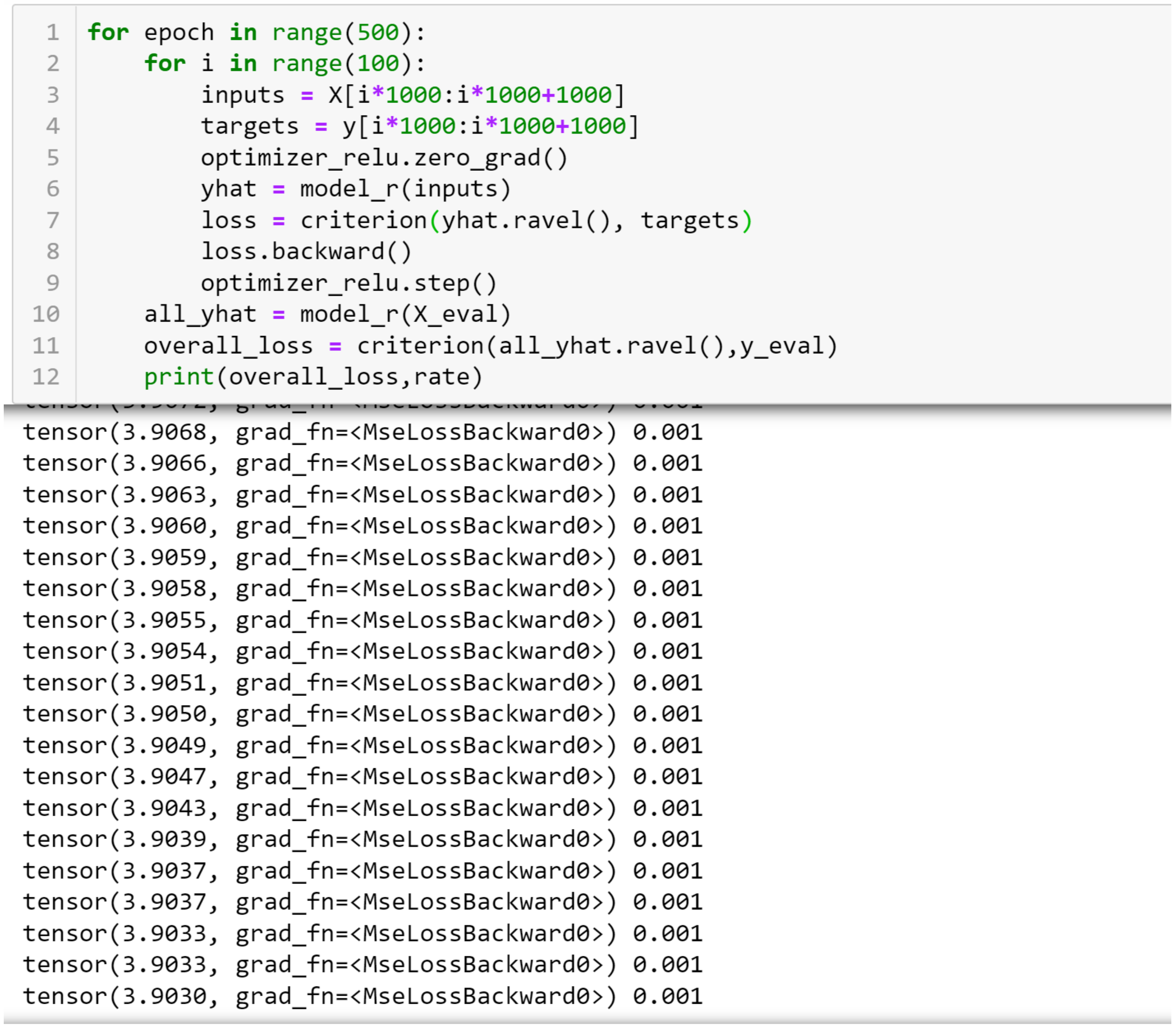
```python
class Net_relu(nn.Module):
    def __init__(self):
        super(Net_relu, self).__init__()
        self.L1 = nn.Linear(10,880,bias=False)
        self.L2 = nn.Linear(880,1,bias=False)
        self.act = nn.ReLU()

    def forward(self, x):
        l1 = self.L1(x)
        l2 = self.act(l1)
        return self.L2(l2)
```

```python
model_r = Net_relu()
```

```python
optimizer_relu = torch.optim.SGD(model_r.parameters(), lr=.001, momentum=0.1)
```

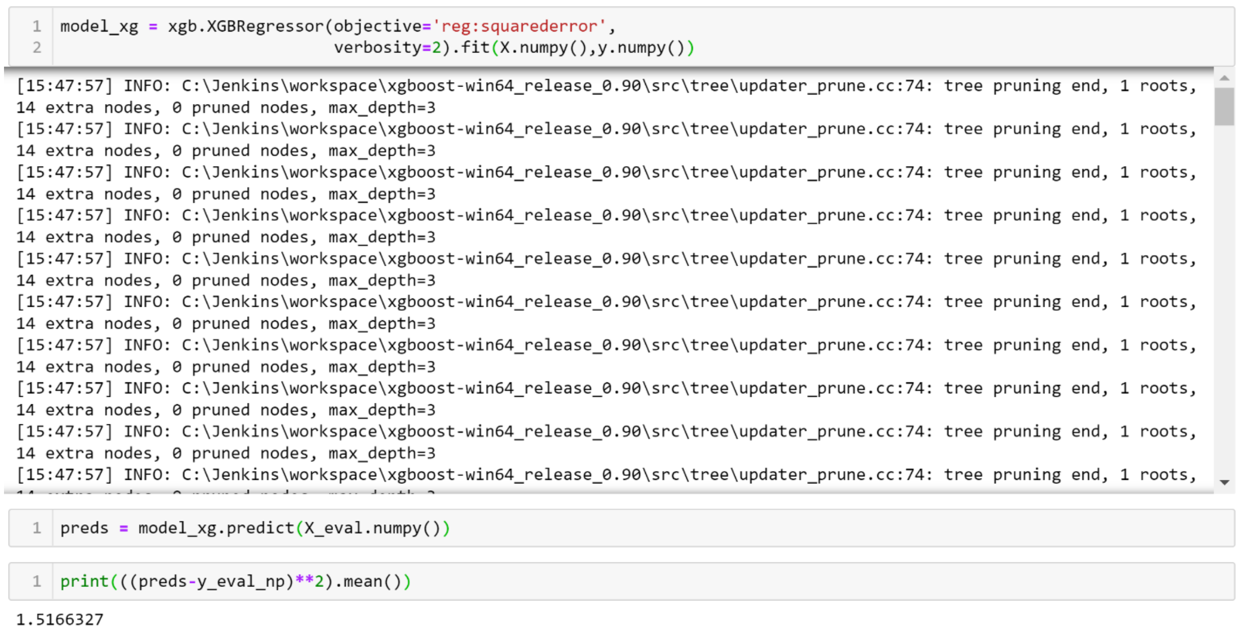
Now let’s train the model.



```python
for epoch in range(500):
    for i in range(100):
        inputs = X[i*1000:i*1000+1000]
        targets = y[i*1000:i*1000+1000]
        optimizer_relu.zero_grad()
        yhat = model_r(inputs)
        loss = criterion(yhat.ravel(), targets)
        loss.backward()
        optimizer_relu.step()
    all_yhat = model_r(X_eval)
    overall_loss = criterion(all_yhat.ravel(),y_eval)
    print(overall_loss,rate)
```

```
tensor(3.9068, grad_fn=<MseLossBackward0>) 0.001
tensor(3.9066, grad_fn=<MseLossBackward0>) 0.001
tensor(3.9063, grad_fn=<MseLossBackward0>) 0.001
tensor(3.9060, grad_fn=<MseLossBackward0>) 0.001
tensor(3.9059, grad_fn=<MseLossBackward0>) 0.001
tensor(3.9058, grad_fn=<MseLossBackward0>) 0.001
tensor(3.9055, grad_fn=<MseLossBackward0>) 0.001
tensor(3.9054, grad_fn=<MseLossBackward0>) 0.001
tensor(3.9051, grad_fn=<MseLossBackward0>) 0.001
tensor(3.9050, grad_fn=<MseLossBackward0>) 0.001
tensor(3.9049, grad_fn=<MseLossBackward0>) 0.001
tensor(3.9047, grad_fn=<MseLossBackward0>) 0.001
tensor(3.9043, grad_fn=<MseLossBackward0>) 0.001
tensor(3.9039, grad_fn=<MseLossBackward0>) 0.001
tensor(3.9037, grad_fn=<MseLossBackward0>) 0.001
tensor(3.9037, grad_fn=<MseLossBackward0>) 0.001
tensor(3.9033, grad_fn=<MseLossBackward0>) 0.001
tensor(3.9033, grad_fn=<MseLossBackward0>) 0.001
tensor(3.9030, grad_fn=<MseLossBackward0>) 0.001
```

The MSE score does drop for the RELU model as training progresses, but it is significantly higher than the MSE score for our custom activation function. So far, the new contender is stacking up well. Let’s try comparing our score against a completely different algorithm. XGboost has been empirically shown to be a phenomenal algorithm for learning arbitrary relationships, and it has been successfully used as the winning algorithm in a variety of machine learning competitions. Let’s compare our algorithm against XGboost.



```
model_xg = xgb.XGBRegressor(objective='reg:squarederror',
                            verbosity=2).fit(X.numpy(),y.numpy())
```

```
[15:47:57] INFO: C:\Jenkins\workspace\xgboost-win64_release_0.90\src\tree\updater_prune.cc:74: tree pruning end, 1 roots,
14 extra nodes, 0 pruned nodes, max_depth=3
[15:47:57] INFO: C:\Jenkins\workspace\xgboost-win64_release_0.90\src\tree\updater_prune.cc:74: tree pruning end, 1 roots,
14 extra nodes, 0 pruned nodes, max_depth=3
[15:47:57] INFO: C:\Jenkins\workspace\xgboost-win64_release_0.90\src\tree\updater_prune.cc:74: tree pruning end, 1 roots,
14 extra nodes, 0 pruned nodes, max_depth=3
[15:47:57] INFO: C:\Jenkins\workspace\xgboost-win64_release_0.90\src\tree\updater_prune.cc:74: tree pruning end, 1 roots,
14 extra nodes, 0 pruned nodes, max_depth=3
[15:47:57] INFO: C:\Jenkins\workspace\xgboost-win64_release_0.90\src\tree\updater_prune.cc:74: tree pruning end, 1 roots,
14 extra nodes, 0 pruned nodes, max_depth=3
[15:47:57] INFO: C:\Jenkins\workspace\xgboost-win64_release_0.90\src\tree\updater_prune.cc:74: tree pruning end, 1 roots,
14 extra nodes, 0 pruned nodes, max_depth=3
[15:47:57] INFO: C:\Jenkins\workspace\xgboost-win64_release_0.90\src\tree\updater_prune.cc:74: tree pruning end, 1 roots,
14 extra nodes, 0 pruned nodes, max_depth=3
[15:47:57] INFO: C:\Jenkins\workspace\xgboost-win64_release_0.90\src\tree\updater_prune.cc:74: tree pruning end, 1 roots,
14 extra nodes, 0 pruned nodes, max_depth=3
[15:47:57] INFO: C:\Jenkins\workspace\xgboost-win64_release_0.90\src\tree\updater_prune.cc:74: tree pruning end, 1 roots,
14 extra nodes, 0 pruned nodes, max_depth=3
[15:47:57] INFO: C:\Jenkins\workspace\xgboost-win64_release_0.90\src\tree\updater_prune.cc:74: tree pruning end, 1 roots,
```

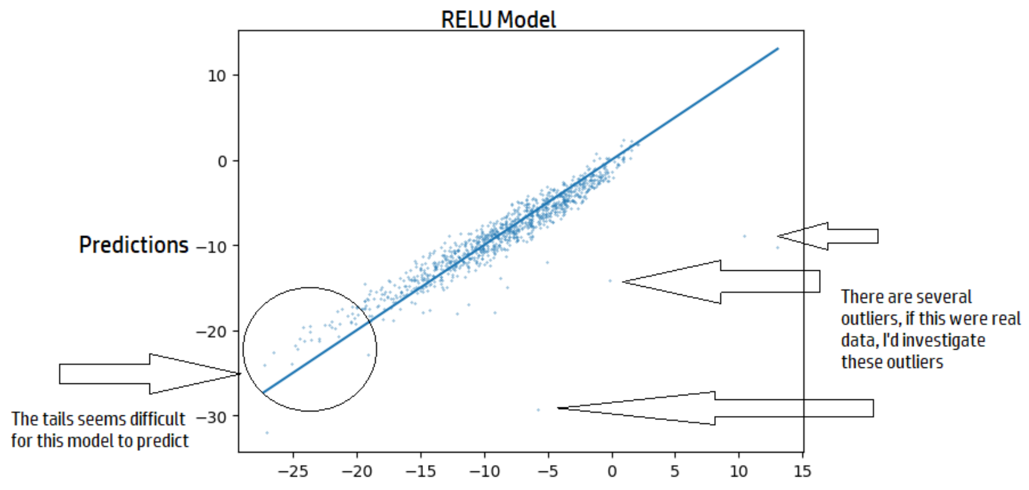
```
preds = model_xg.predict(X_eval.numpy())
```

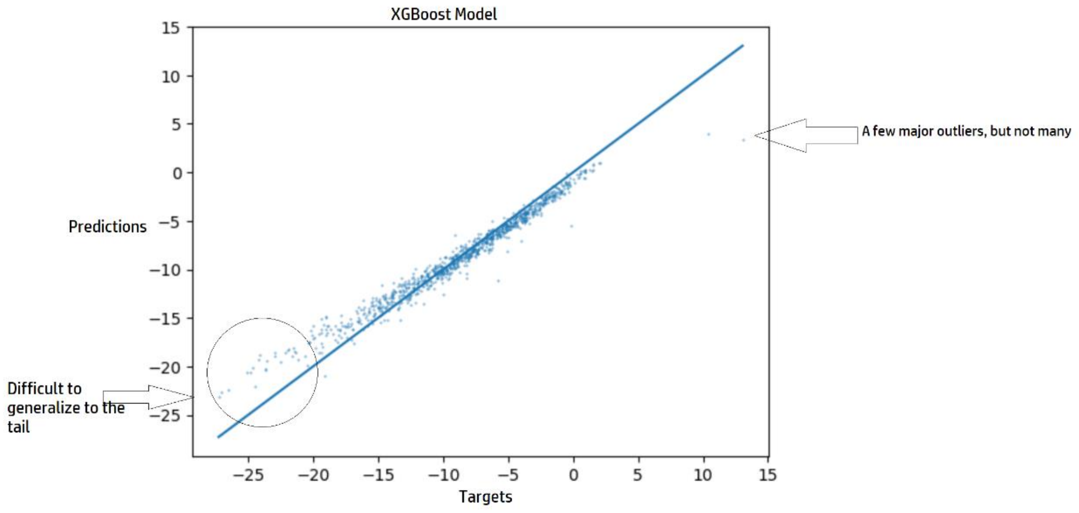
```
print(((preds-y_eval_np)**2).mean())
```

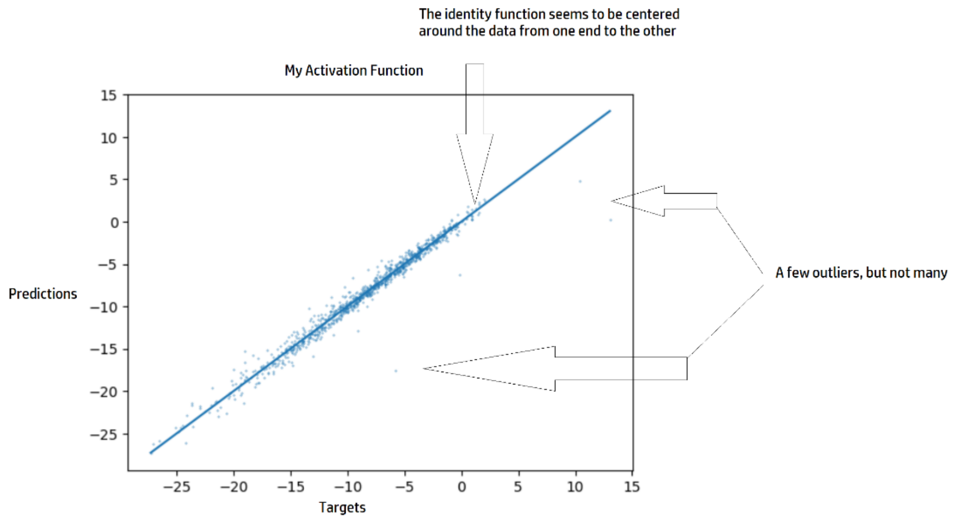
```
1.5166327
```

The XGboost model beat the RELU neural network, but it still falls short of beating our new custom activation function. This indicates that our proposed activation function seems to be a great candidate for further investigation, and we should be especially interested in applying it to real world problems to see how it performs on non-synthetic data.

To help visualize the results, I have created a scatter plot of predictions vs targets for each model.

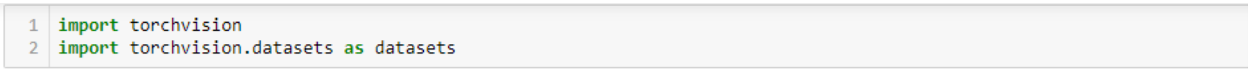




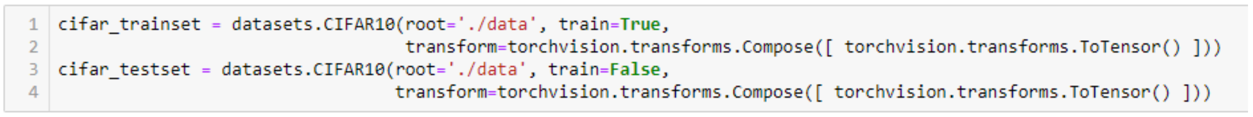


Now that we have some strong evidence that this new activation function seems to work very well, let's try using this activation function on a real-world problem. I will train two CNN models on the CIFAR-10 data set, one with our new activation function, and the second with the RELU activation function.

First let's get the data:

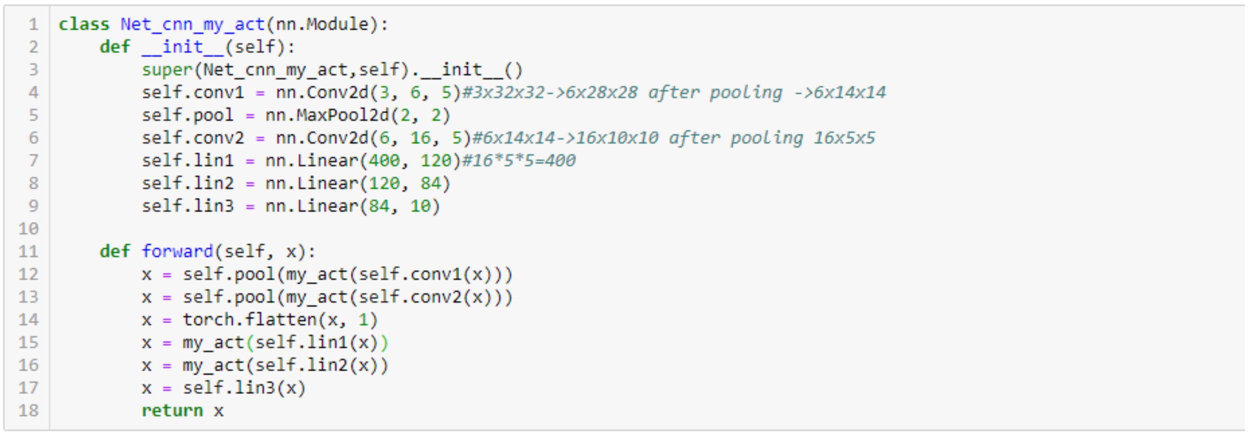


```
import torchvision
import torchvision.datasets as datasets
```



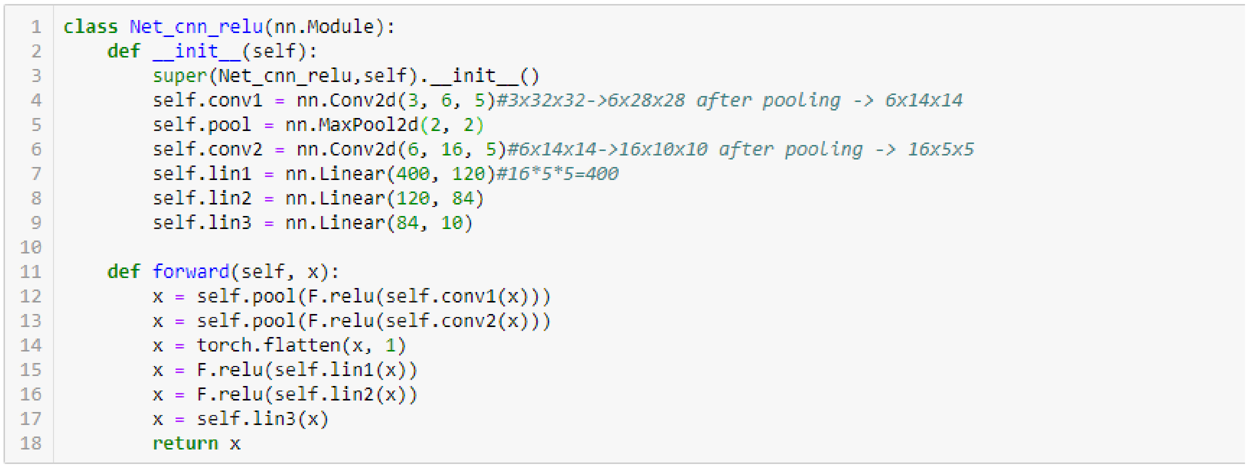
```
cifar_trainset = datasets.CIFAR10(root='./data', train=True,
                                  transform=torchvision.transforms.Compose([ torchvision.transforms.ToTensor() ]))
cifar_testset = datasets.CIFAR10(root='./data', train=False,
                                 transform=torchvision.transforms.Compose([ torchvision.transforms.ToTensor() ]))
```

And let's define our models:



```
class Net_cnn_my_act(nn.Module):
    def __init__(self):
        super(Net_cnn_my_act,self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)#3x32x32->6x28x28 after pooling ->6x14x14
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)#6x14x14->16x10x10 after pooling 16x5x5
        self.lin1 = nn.Linear(400, 120)#16*5*5=400
        self.lin2 = nn.Linear(120, 84)
        self.lin3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(my_act(self.conv1(x)))
        x = self.pool(my_act(self.conv2(x)))
        x = torch.flatten(x, 1)
        x = my_act(self.lin1(x))
        x = my_act(self.lin2(x))
        x = self.lin3(x)
        return x
```



```
class Net_cnn_relu(nn.Module):
    def __init__(self):
        super(Net_cnn_relu,self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)#3x32x32->6x28x28 after pooling -> 6x14x14
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)#6x14x14->16x10x10 after pooling -> 16x5x5
        self.lin1 = nn.Linear(400, 120)#16*5*5=400
        self.lin2 = nn.Linear(120, 84)
        self.lin3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)
        x = F.relu(self.lin1(x))
        x = F.relu(self.lin2(x))
        x = self.lin3(x)
        return x
```

Now let’s instantiate our model, get our data loader, optimizer, and loss function:

```python
my_cnn = Net_cnn_my_act()
```

```python
train_loader = torch.utils.data.DataLoader(cifar_trainset,batch_size=64, shuffle=True)
```

```python
my_cnn_optimizer = torch.optim.SGD(my_cnn.parameters(), lr=.0015, momentum=0.2)
cnn_loss = torch.nn.CrossEntropyLoss()
```

```python
test_loader = torch.utils.data.DataLoader(cifar_testset,batch_size=10000, shuffle=False)
```

And let’s train the model:

```python
for epoch in range(50):
    for (batch_idx, batch) in enumerate(train_loader):
        (X, Y) = batch
        my_cnn_optimizer.zero_grad()
        preds = my_cnn(X)
        loss_val = cnn_loss(preds, Y)
        loss_val.backward()
        my_cnn_optimizer.step()
        print(batch_idx,end='\r')
    for d in test_loader:
        (X, Y) = d
        preds = my_cnn(X)
        loss_val = cnn_loss(preds, Y)
        print(loss_val)
```

```
tensor(1.2689, grad_fn=<NllLossBackward0>)
tensor(1.3262, grad_fn=<NllLossBackward0>)
tensor(1.2512, grad_fn=<NllLossBackward0>)
tensor(1.2766, grad_fn=<NllLossBackward0>)
tensor(1.2617, grad_fn=<NllLossBackward0>)
tensor(1.2371, grad_fn=<NllLossBackward0>)
tensor(1.2289, grad_fn=<NllLossBackward0>)
tensor(1.2364, grad_fn=<NllLossBackward0>)
tensor(1.2617, grad_fn=<NllLossBackward0>)
tensor(1.3293, grad_fn=<NllLossBackward0>)
tensor(1.2494, grad_fn=<NllLossBackward0>)
tensor(1.2491, grad_fn=<NllLossBackward0>)
tensor(1.2371, grad_fn=<NllLossBackward0>)
tensor(1.3231, grad_fn=<NllLossBackward0>)
tensor(1.2316, grad_fn=<NllLossBackward0>)
tensor(1.2492, grad_fn=<NllLossBackward0>)
tensor(1.2104, grad_fn=<NllLossBackward0>)
tensor(1.3047, grad_fn=<NllLossBackward0>)
tensor(1.2121, grad_fn=<NllLossBackward0>)
```

And let’s instantiate our RELU model and train it with an identical optimizer:

```python
relu_cnn = Net_cnn_relu()
```

```python
relu_cnn_optimizer = torch.optim.SGD(relu_cnn.parameters(), lr=.0015, momentum=0.2)
```

```python
for epoch in range(0, 50):
    for (batch_idx, batch) in enumerate(train_loader):
        (X, Y) = batch
        relu_cnn_optimizer.zero_grad()
        preds = relu_cnn(X)
        loss_val = cnn_loss(preds, Y)
        loss_val.backward()
        relu_cnn_optimizer.step()
        print(batch_idx,end='\r')
    for d in test_loader:
        (X, Y) = d
        preds = relu_cnn(X)
        loss_val = cnn_loss(preds, Y)
        print(loss_val)
```

```
tensor(1.6560, grad_fn=<NllLossBackward0>)
tensor(1.6711, grad_fn=<NllLossBackward0>)
tensor(1.6299, grad_fn=<NllLossBackward0>)
tensor(1.6908, grad_fn=<NllLossBackward0>)
tensor(1.6081, grad_fn=<NllLossBackward0>)
tensor(1.6020, grad_fn=<NllLossBackward0>)
tensor(1.5772, grad_fn=<NllLossBackward0>)
tensor(1.6781, grad_fn=<NllLossBackward0>)
tensor(1.6234, grad_fn=<NllLossBackward0>)
tensor(1.5264, grad_fn=<NllLossBackward0>)
tensor(1.5996, grad_fn=<NllLossBackward0>)
tensor(1.5667, grad_fn=<NllLossBackward0>)
tensor(1.6917, grad_fn=<NllLossBackward0>)
tensor(1.4675, grad_fn=<NllLossBackward0>)
tensor(1.6357, grad_fn=<NllLossBackward0>)
tensor(1.4622, grad_fn=<NllLossBackward0>)
tensor(1.4443, grad_fn=<NllLossBackward0>)
tensor(1.4529, grad_fn=<NllLossBackward0>)
tensor(1.4633, grad_fn=<NllLossBackward0>)
```

We can see immediately from the loss values during training that our activation function seems to outperform RELU, but let’s compare the accuracy:

```python
for d in test_loader:
    (X, Y) = d
    preds = my_cnn(X)
    acc_custom = ((preds.argmax(axis=1)-Y)==0).sum()/Y.size().numel()
```

```python
for d in test_loader:
    (X, Y) = d
    preds = relu_cnn(X)
    acc_relu = ((preds.argmax(axis=1)-Y)==0).sum()/Y.size().numel()
```

```python
print('Custom Activation accuracy:\t{}'.format(round(acc_custom.item(),4)))
print('RELU Activation accuracy:\t{}'.format(round(acc_relu.item(),4)))
```

```
Custom Activation accuracy:     0.5826
RELU Activation accuracy:       0.4784
```

It certainly appears that our novel activation function has great potential in a variety of different problem statements. Further tests in more applications will be needed to determine exactly how useful this new activation function, but the initial results certainly warrant more investigation.