Name: Corbin Charpentier SID: 29289

UW

Homework 2

Module

1 Landscape of Top Eigenvector Problem

Consider the following optimization problem for computing the top eigenvector of a positive semidefinite matrix $M \in \mathbb{R}^{d\times d}$:

$$\min_{x\in\mathbb{R}^d} f(x) = \frac{1}{4}\left\|xx^\top - M\right\|_F^2.$$

Let the eigen-decomposition of M be $M = \sum_{i=1}^d \lambda_i v_i v_i^\top$ where $\{\lambda_i\}_{i=1}^d$ are eigenvalues and $\{v_i\}_{i=1}^d$ are the corresponding eigenvectors. Assume $\lambda_1 > \lambda_2 \geq \lambda_3 \geq \cdots \lambda_d \geq 0$. Prove that all saddle points and local maxima are strict, i.e., except the global minima, the Hessian matrices of all other first-order stationary points have a negative eigenvalue.

First we calculate the gradient of f. We will use that to identify stationary points. We rewrite f in a way that makes allows the partial derivative to be computed.

$$f(x) = \frac{1}{4}\sum_{i=1}^{d}\sum_{j=1}^{d}(x_ix_j - M_{ij})^2$$

So, using the chain rule (disclosure: classmate, Roger Wong, helped me figure out this derivative)

$$\begin{aligned}\frac{\partial f}{\partial x_i} &= \frac{1}{4}\sum_{j\neq i,j=1}^{d} 2x_j\left(x_ix_j - M_{ij}\right) + \frac{1}{4}\sum_{j\neq i,j=1}^{d} 2x_j\left(x_ix_j - M_{ji}\right) + \frac{1}{4}(2)(2x_i)\left(x_i^2 - M_{ii}\right)\\ &= \sum_{j=1}^{d} x_j\left(x_ix_j - M_{ij}\right)\end{aligned}$$

Finding the partial derivative of each x_i leads directly to the the gradient:

$$\nabla f(x) = \begin{bmatrix} \sum_{j=1}^d x_j\left(x_1x_j - M_{1j}\right)\\ \sum_{j=1}^d x_j\left(x_2x_j - M_{2j}\right)\\ \cdot\\ \cdot\\ \cdot\\ \sum_{j=1}^d x_j\left(x_dx_j - M_{dj}\right)\end{bmatrix} = \|x\|_2^2 x - Mx$$

The stationary points exist where $\nabla f(x) = 0$, which implies

$$\implies \|x\|_2^2 x = Mx$$

Which is the definition of an eigenvector, where x is an eigenvector with corresponding eigenvalue $\|x\|_2^2$. So, stationary points of f occur when x is a unit eigenvector of M, hence we divide by the norm of x, yielding $x = \sqrt{\lambda}v$. Using identities on the matrix calculus wikipedia page, the Hessian of f is

$$\nabla^2 f(x) = 2xx^2 + \|x\|_2^2 I - M$$

where I is the identity matrix. Plugging the stationary point in

$$\begin{aligned}\nabla^2 f(x) &= 2(\sqrt{\lambda}v)(\sqrt{\lambda}v)^T + (\sqrt{\lambda}v)^T(\sqrt{\lambda}v)I - M\\ &= 2\lambda vv^t + \lambda v^T vI - M\\ &= \lambda(2vv^T + v^T vI) - M\end{aligned}$$

Since λ will always be less than the largest eigenvalue in M, $\lambda_{min}\{\nabla^2 f(x)\} < 0$

Problem 2.1 Show $L(w_t) \to 0$ as $t \to \infty$

$L(w)$ can be rewritten as

$$L(w) = \frac{1}{2n}\|X^T w - y\|_2^2$$

We derive the dynamics of the $L(w)$ using the general formula given in class

We can conclude that $\lambda_{min}(H(t)) > 0$

Problem 2.2 Show that w is always in the span of $(x_1, ..., x_n)$

We need to show that w is a linear combination of the column vectors of X, i.e.

$$w = a_1 x_1 + ... + a_n x_n$$

Observe that each row of $H(t)$ (defined in problem 2.1) represents a linear combination of x_i up to the nth dimension (where $n \leq d$).
Let $z = (X^T w - y)$. It is given that $\frac{dw_t}{dt} = -\nabla L(w_t)$, which implies

$$\begin{aligned} w_{t+1} &= w_t + z^T H(0) z \\ &= \sum_{i,j=1}^{n} [H(0)]_{ij} z_i z_j \end{aligned}$$

So, each update of w (start with the 0 vector, which is indeed in the span of X) adds a linear combination of X to the previous value of w. Since the sum of two linear combinations X is also a linear combination of X, w is always in the span of X.

Problem 2.3

Assume, given the constraints of the problem, that w_t minimizes $L(W)$ but not $\|w\|_2^2$ as $t \to \infty$ (we know $L(W)$ is minimized from the result of problem 2.1). Since w is in the span of X, this can't be true: w must also shrink is $L(W)$ shrinks, given $y_i = x_i^T w$.

Problem 3.1

We are given

$$k(x, x') = x^\top x' \cdot \mathbb{E}_w \left[\sigma'(w_1) \sigma' \left(w_1 x^\top x' + w_2 \sqrt{1 - (x^\top x')^2} \right) \right]$$

Letting $x^T x = \cos\theta$ gives

$$k(x, x') = \cos\theta \cdot \mathbb{E}_w [\sigma'(w_1) \sigma'(w_1 \cos\theta + w_2 \sin\theta)]$$

Now, let $w_1^* = w_1$ and $w_2^* = w_1 \cos\theta + w_2 \sin\theta$. Solving for w_1 yields $w_1 = \frac{w_2^* - w_2 \sin\theta}{\cos\theta}$, which we plug into k as follows:

$$\begin{aligned} k(x, x') &= \cos\theta \cdot \mathbb{E}_w [\sigma'(w_1) \sigma'(w_1 \cos\theta + w_2 \sin\theta)] \\ &= \cos\theta \cdot \mathbb{E}_w \left[\sigma'(w_1^*) \sigma' \left(\frac{w_2^* - w_2 \sin\theta}{\cos\theta} \cos\theta + w_2 \sin\theta \right) \right] \\ &= \cos\theta \cdot \mathbb{E}_w [\sigma'(w_1^*) \sigma'(w_2^*)] \end{aligned}$$

Since $\|x\|, \|x'\| = 1$, both $w^T x^{(\prime)}$ and w_i are distributed $N(0, I)$ (and are therefore rotationally invariant), so:

$$\begin{aligned} k(x, x') &= \cos\theta \cdot \mathbb{E}_w [\sigma'(w_1^*) \sigma'(w_2^*)] \\ &= x^T x \cdot \mathbb{E}_w [\sigma'(w_1^*) \sigma'(w_2^*)] = x^T x \cdot \mathbb{E}_w \left[\sigma'(w^T x) \sigma'(w^T x') \right] \end{aligned}$$

Problem 3.2

Start with

$$
\begin{aligned}
k\left(x, x'\right) &= \cos\theta \cdot \mathbb{E}_w\left[\sigma'\left(w_1\right)\sigma'\left(w_1\cos\theta + w_2\sin\theta\right)\right] \\
&= \cos\theta \cdot \int_{-\infty}^{\infty} \sigma'\left(w_1\right)\sigma'\left(w_1\cos\theta + w_2\sin\theta\right)P(w)dw \\
&= \frac{\cos\theta}{2\pi} \cdot \int_{0}^{\infty} w_1(w_1\cos\theta + w_2\sin\theta)(e^{\frac{-1}{2}(w_1^2+w_2^2)})dw_1dw_2
\end{aligned}
$$

Now, letting $u = w_1$ and $v = w_1\cos\theta + w_2\sin\theta$...

(Bound by $\pi/2$ since this is the upper range of the domain when $\|x\| = 1$)

$$
\begin{aligned}
&= \frac{\cos\theta}{2\pi} \cdot \int_{0}^{\pi/2} uve^{\frac{-1}{2}(u^2+\left(\frac{v-u\cos\theta}{\sin\theta}\right)^2)}dudv \\
&= \frac{\cos\theta}{2\pi} \cdot \int_{0}^{\pi/2} uve^{-1(u^2+v^2-2uv\cos\theta)/2\sin^2\theta}dudv
\end{aligned}
$$

Change to polar coordinates where $u = r\cos\phi, v = r\sin\phi$

$$
= \frac{\cos\theta}{2\pi} \cdot \int_{0}^{\pi/2} r^2\cos\phi\sin\phi \times e^{-1(u^2+v^2-2uv\cos\theta)/2\sin^2\theta}drd\phi
$$

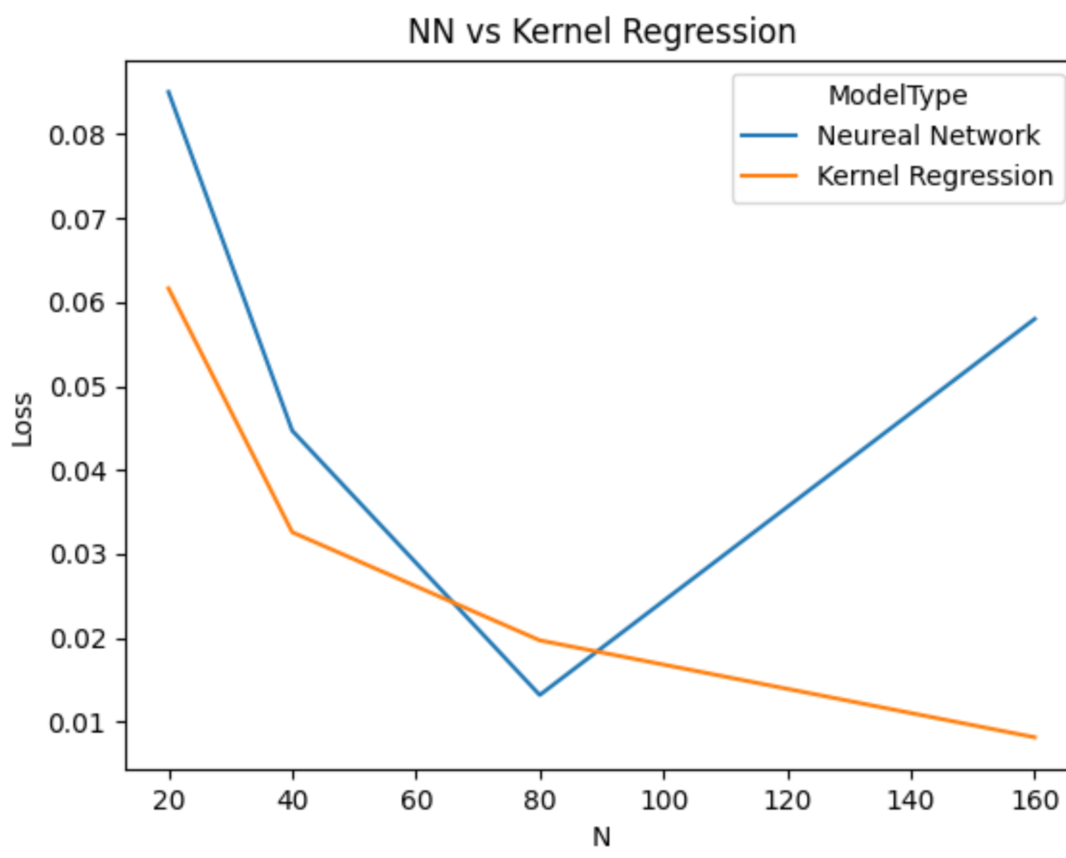
I couldn't figure out how to solve this integral, but I'm sure it evaluates to $\pi - \theta$, so we'd have

$$
\begin{aligned}
k\left(x, x'\right) &= \frac{\cos\theta}{2\pi} \cdot \int_{0}^{\pi/2} r^2\cos\phi\sin\phi \times e^{-1(u^2+v^2-2uv\cos\theta)/2\sin^2\theta}drd\phi \\
&= \frac{\cos\theta}{2\pi} \cdot (\pi - \theta) \\
&= \frac{x^Tx'(\pi - \cos^{-1}(x^Tx'))}{2\pi}.
\end{aligned}
$$

The last step follows from the fact that we initially set $x^Tx' = \cos\theta$

3.3

The neural network generalized less way to the test data, especially as sample size increased. It's likely that the NN starts to overfit (high variance) the training data as sample size increases since we have much fewer parameters than data.



Learning rate was set to 0.2. See code below at the end of this report.

Problem 4.1: Initialization for Leaky ReLU

Expressing z^h recursively, we have $z^h = w^h x^h$ and $x^h = \sigma(z^{h-1})$. We also know that $E(z_i^h) = 0$ the weights are given to be initialized normally. We want to find a $var(z^h)$ such that

$$\text{var}(z^h) = d_h \text{var}(w^h x^h) = d_{h-1} \text{var}(w^{h-1} x^{h-1}) = \cdots = d_1 \text{var}(w^1 x^1)$$

Which implies

$$\text{var}(z^h) = d_h \text{var}(w_{ij}^h) E[(x_i^h)^2]$$

Now we rewrite $E[(x_i^h)^2]$ in terms of variance

$$\begin{aligned}
E[(x_i^h)^2] &= E[(\sigma(z_j^{h-1})^2] \\
&= E[(\max(0, z_j^{h-1}) + \alpha \min(0, z_j^{h-1}))^2] \\
&= E[\max(0, z_j^{h-1})^2 + \max(0, z_j^{h-1}) \times \alpha \mathrm{min}(0, z_j^{h-1}) + \alpha^2 \mathrm{min}(0, z_j^{h-1})^2] \\
&= E[\max(0, z_j^{h-1})^2] + E[\alpha^2 \mathrm{min}(0, z_j^{h-1})^2] \\
&= \int_{-\infty}^{\infty} \max(0, z_j^{h-1})^2 P(z_j^{h-1}) dz_j^{h-1} + \alpha^2 \int_{-\infty}^{\infty} \min(0, z_j^{h-1})^2 P(z_j^{h-1}) dz_j^{h-1} \\
&= \int_{0}^{\infty} (z_j^{h-1})^2 P(z_j^{h-1}) dz_j^{h-1} + \alpha^2 \int_{-\infty}^{0} (z_j^{h-1})^2 P(z_j^{h-1}) dz_j^{h-1} \\
&= \frac{1}{2} \int_{-\infty}^{\infty} (z_j^{h-1})^2 P(z_j^{h-1}) dz_j^{h-1} + \alpha^2 \frac{1}{2} \int_{-\infty}^{\infty} (z_j^{h-1})^2 P(z_j^{h-1}) dz_j^{h-1} \\
&= \frac{1}{2} \mathrm{var}(z_j^{h-1}) + \frac{\alpha^2}{2} \mathrm{var}(z_j^{h-1}) \\
&= \frac{1}{2} \mathrm{var}(z_j^{h-1})(1 + \alpha^2)
\end{aligned}$$

Therefore

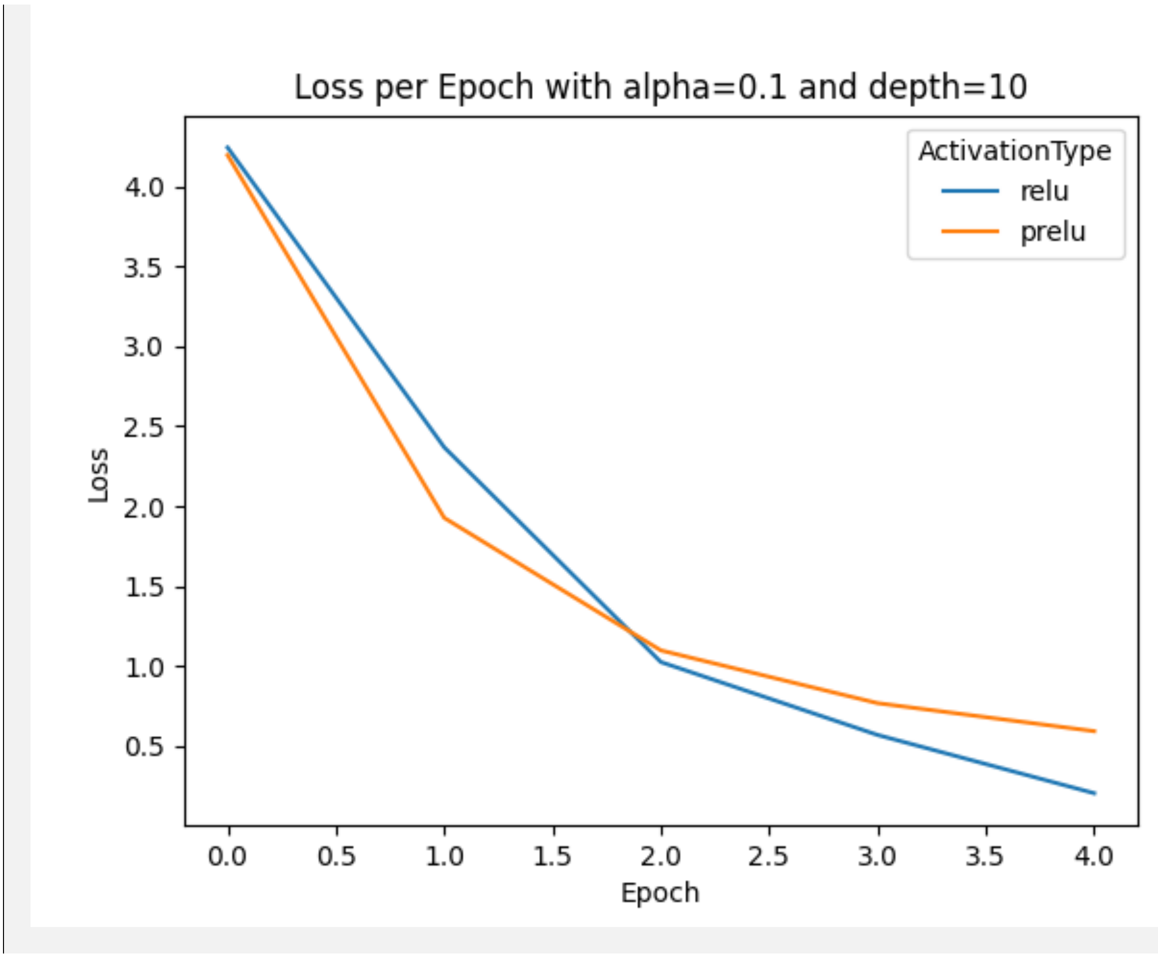
$$\mathrm{var}(z^h) = d_h \mathrm{var}(w_{ij}^h) \frac{1}{2} \mathrm{var}(z_j^{h-1})(1 + \alpha^2)$$

And now solving for $\mathrm{var}(w_{ij}^h)$, where all z has the same variance at all layers

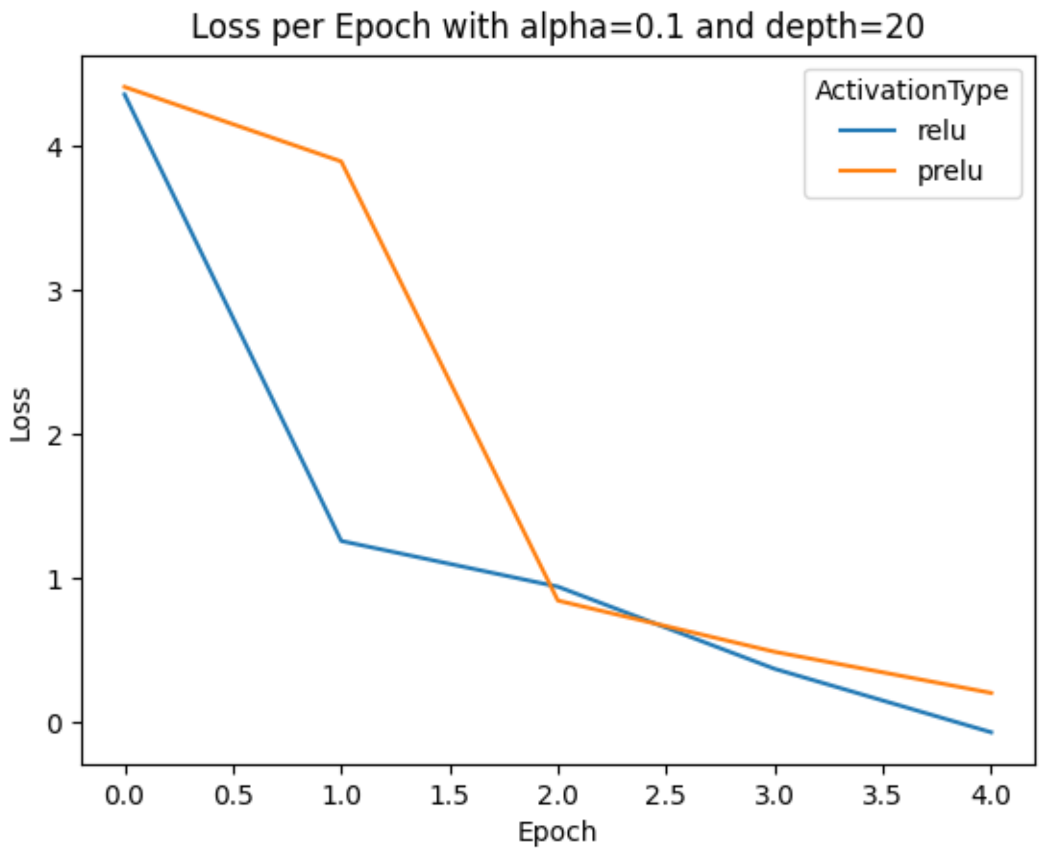
$$\mathrm{var}(w_{ij}^h) = \beta_h = \frac{2}{d_h(1 + \alpha^2)}$$

Problem 4.2

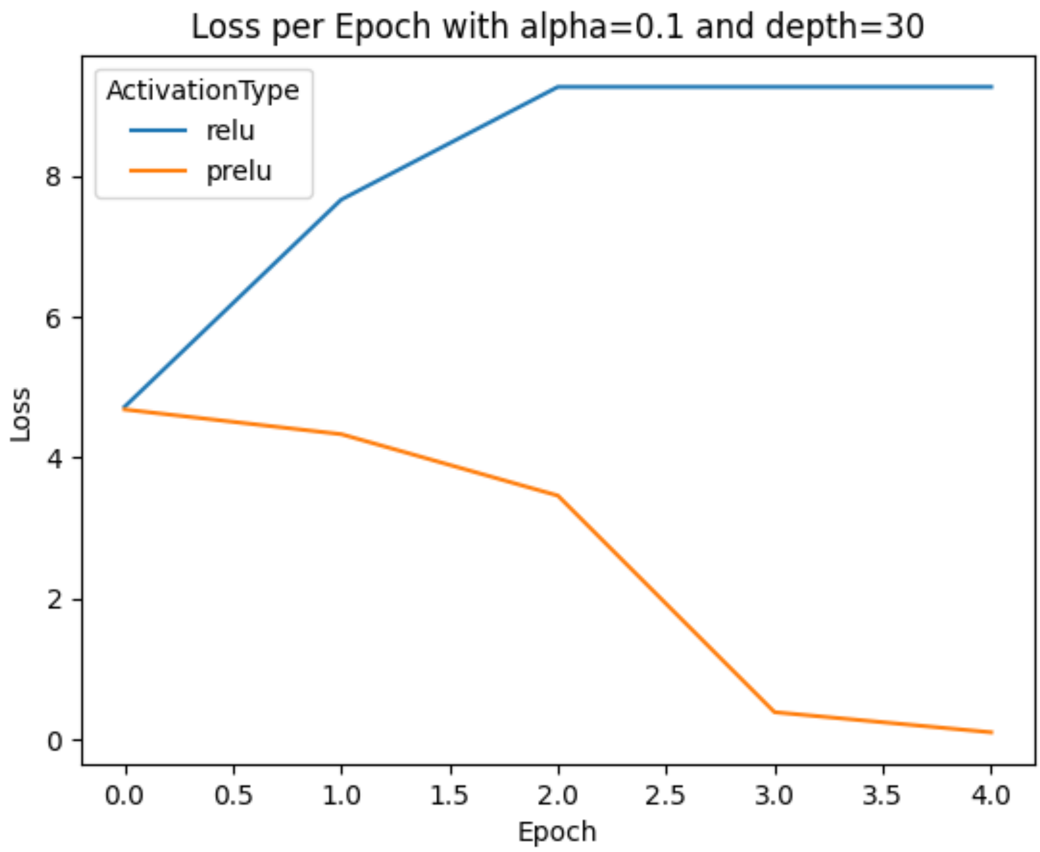
Learning rate for all computations was 0.1. All other hyper-parameters are assigned as specified in the problem statement.

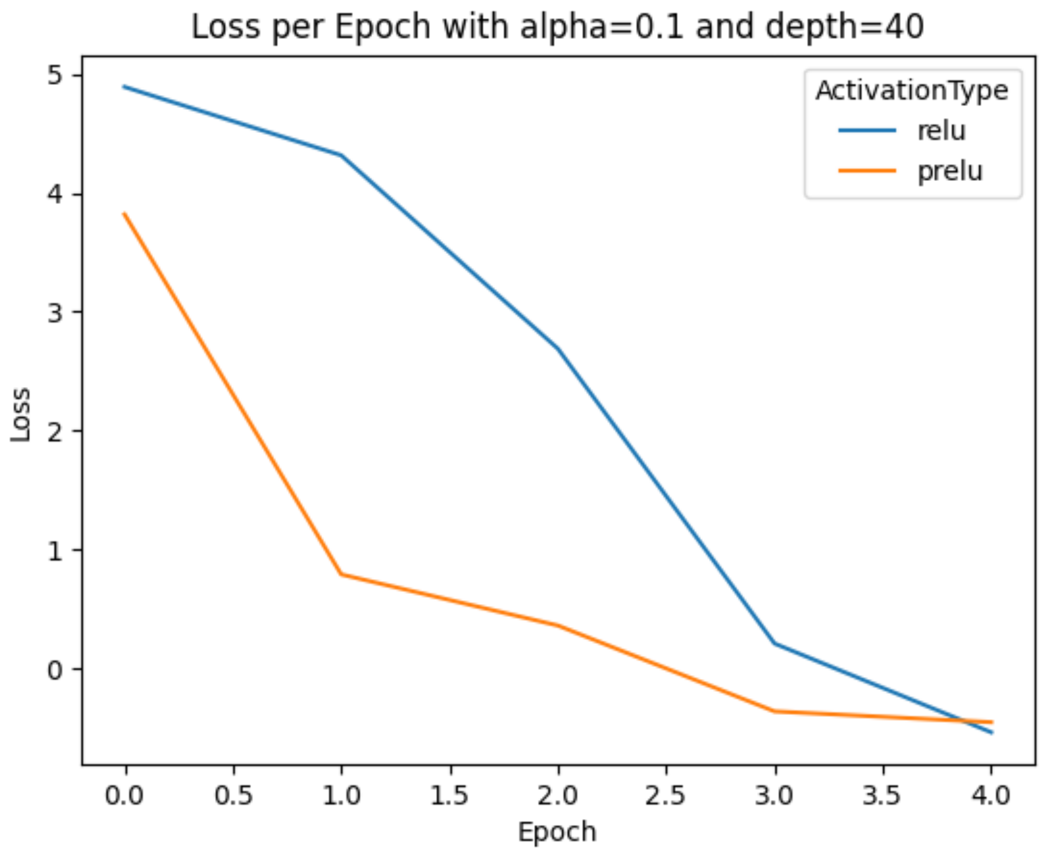


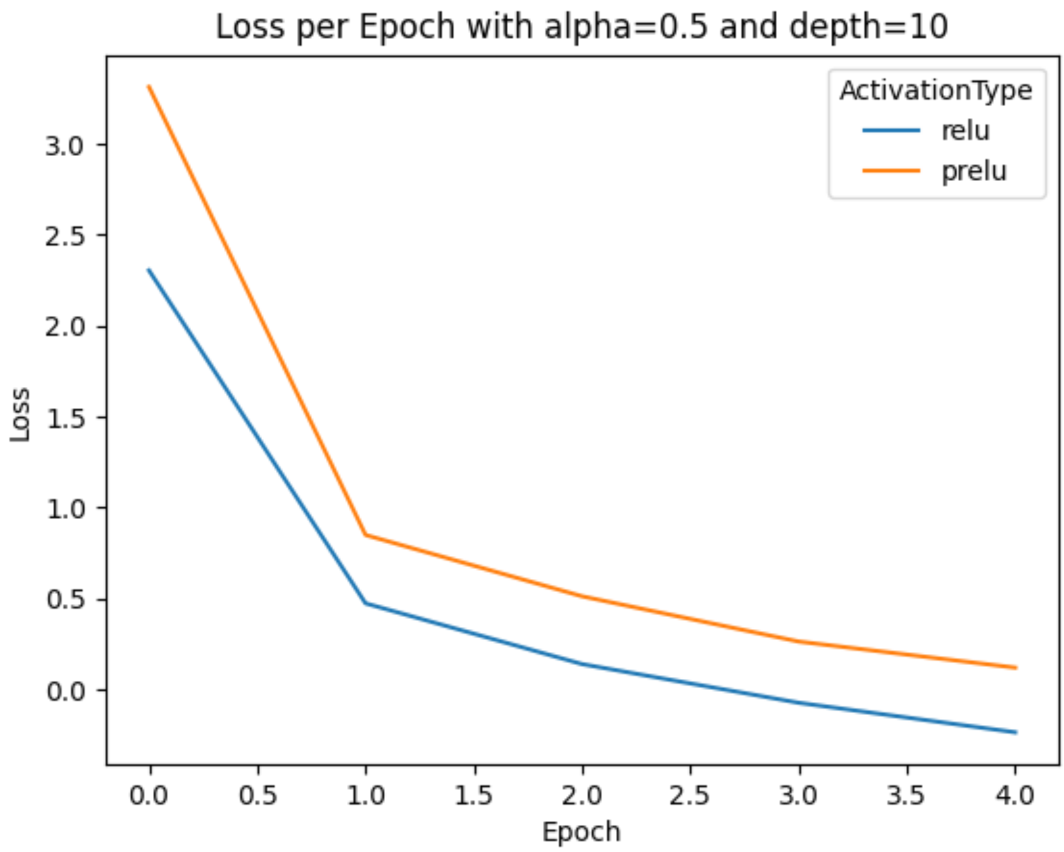
Loss per Epoch with alpha=0.1 and depth=10
ActivationType
relu
prelu
Loss
Epoch
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0



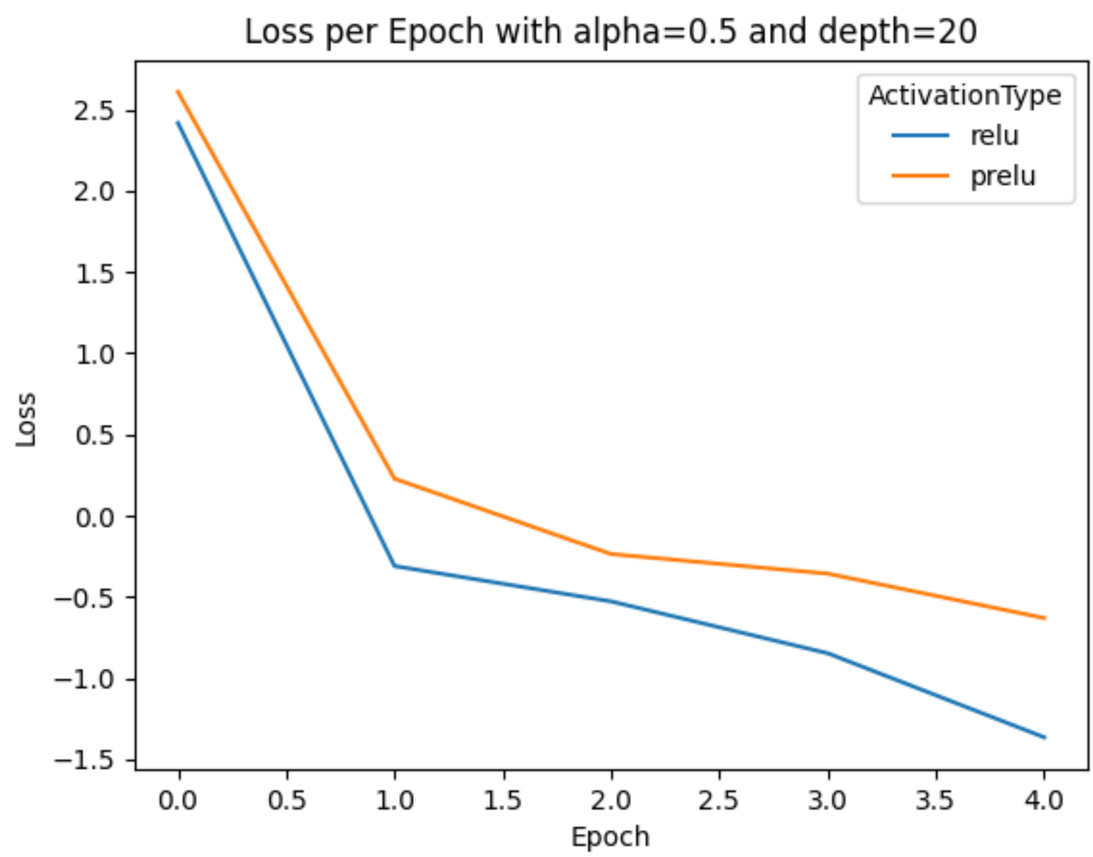
Loss per Epoch with alpha=0.1 and depth=20
ActivationType
relu
prelu
Loss
Epoch
0
1
2
3
4
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0


Loss per Epoch with alpha=0.1 and depth=30
ActivationType
relu
prelu
Loss
Epoch
0
2
4
6
8
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

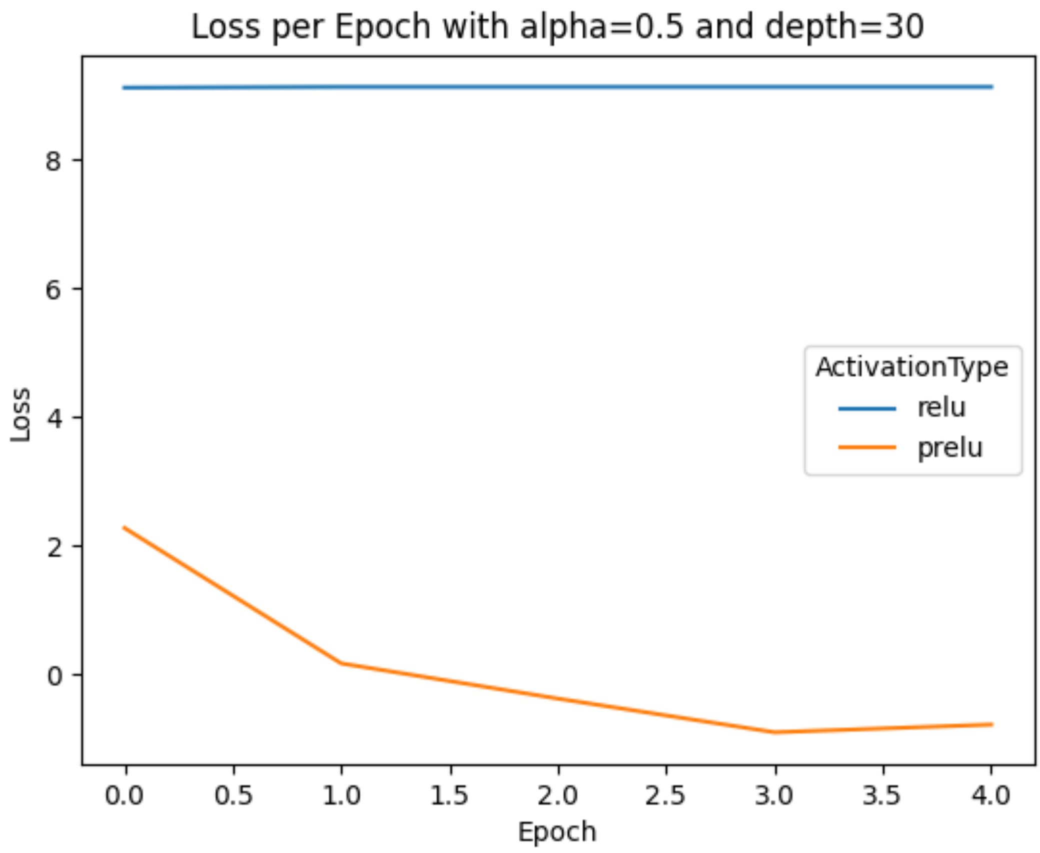

Loss per Epoch with alpha=0.1 and depth=40
ActivationType
relu
prelu
Loss
Epoch
5
4
3
2
1
0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0



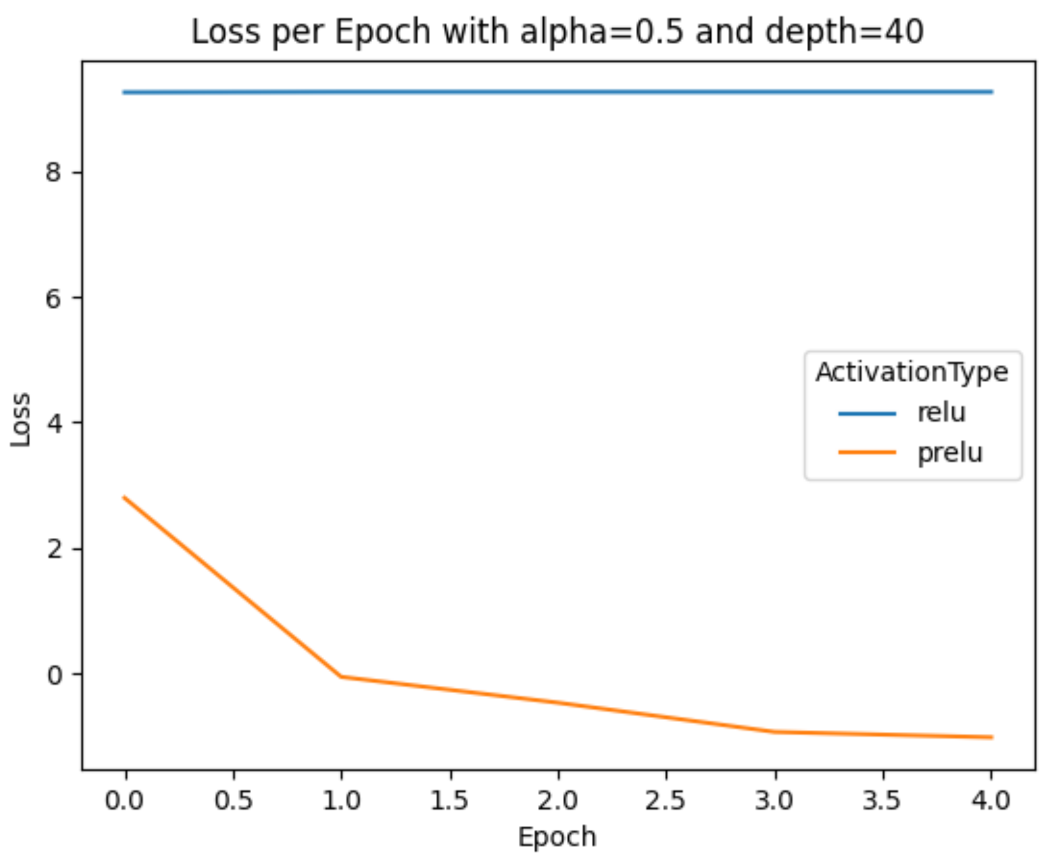
Loss per Epoch with alpha=0.5 and depth=10
ActivationType
relu
prelu
Loss
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Epoch

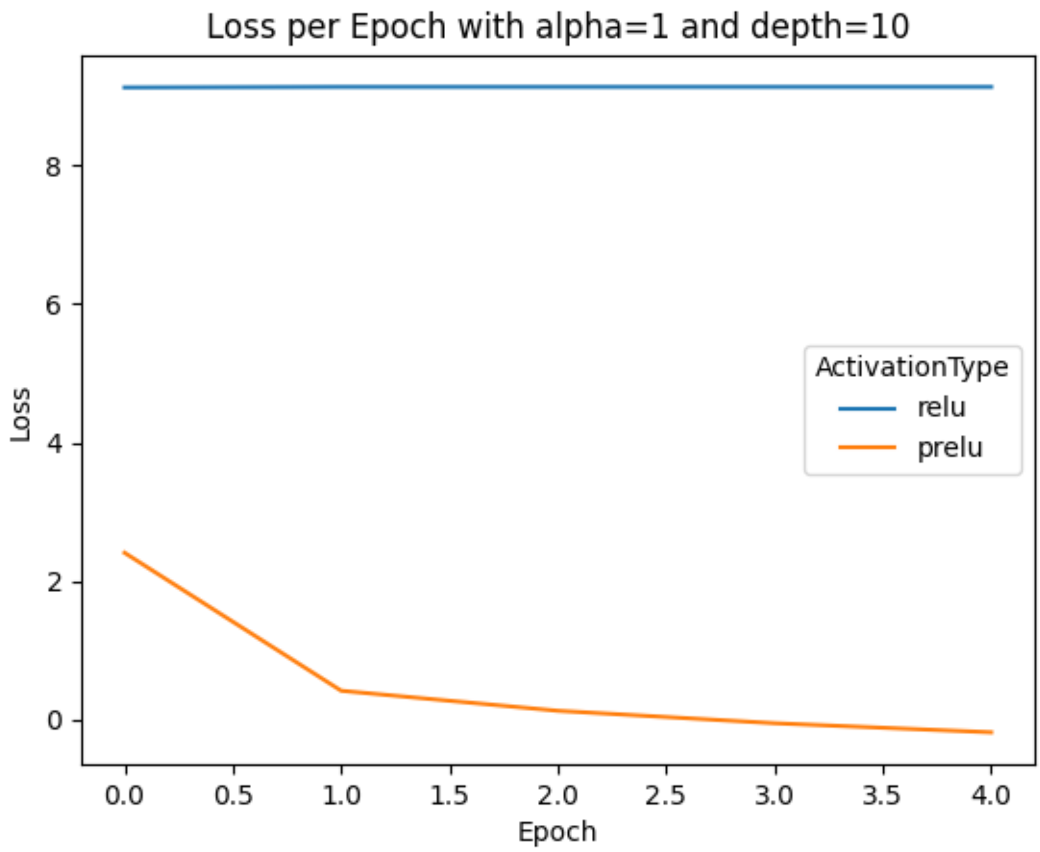


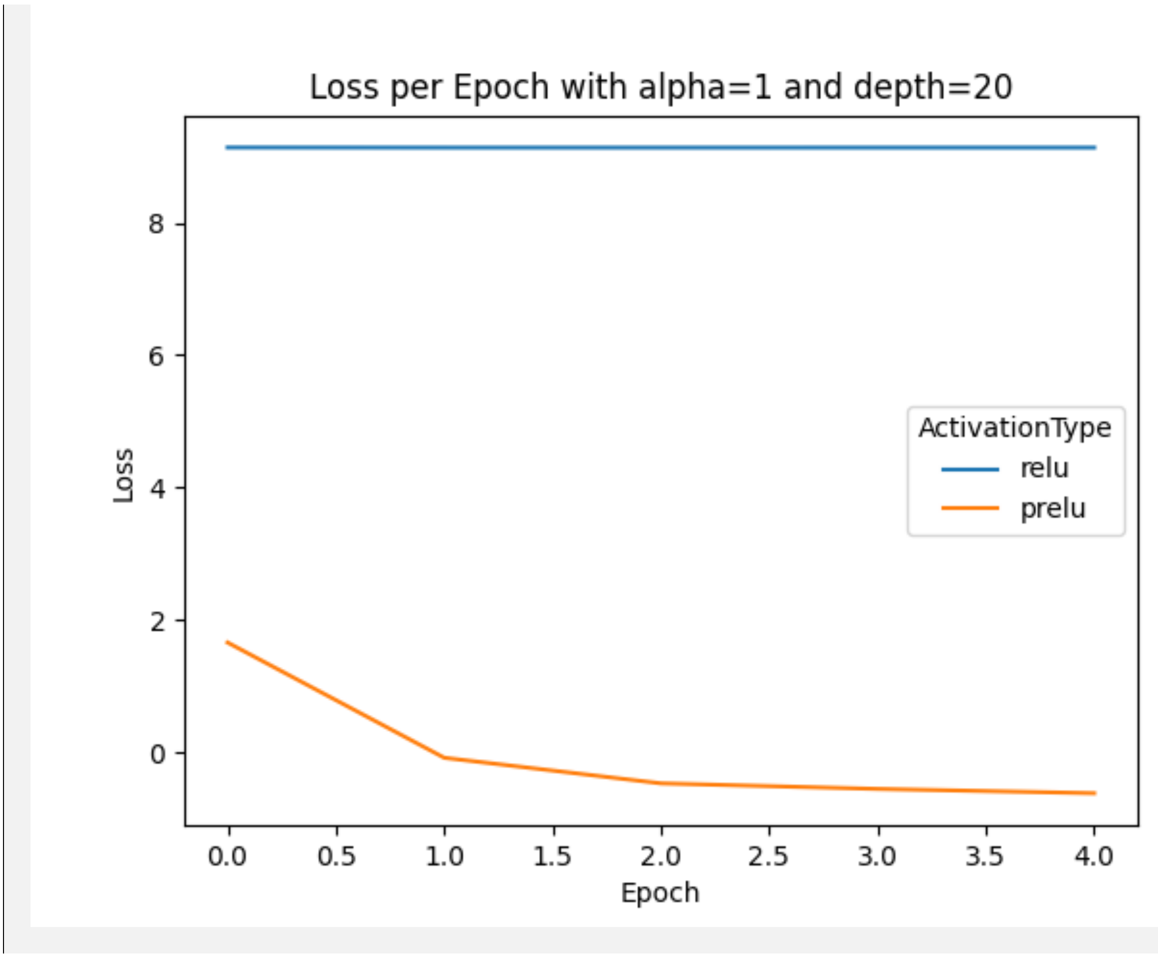
Loss per Epoch with alpha=0.5 and depth=20
ActivationType
relu
prelu
Loss
Epoch
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0



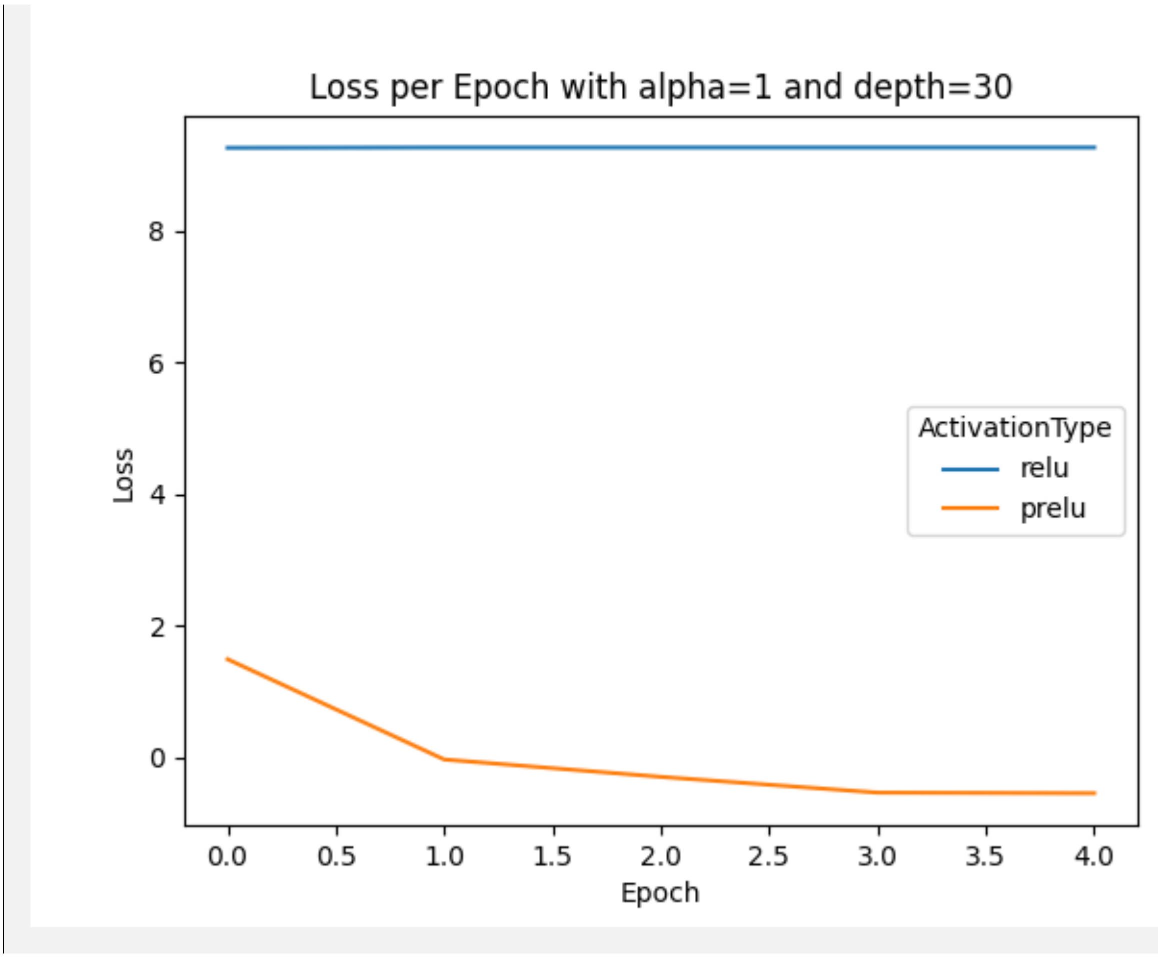
Loss per Epoch with alpha=0.5 and depth=30
Loss
Epoch
ActivationType
relu
prelu
0
2
4
6
8
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

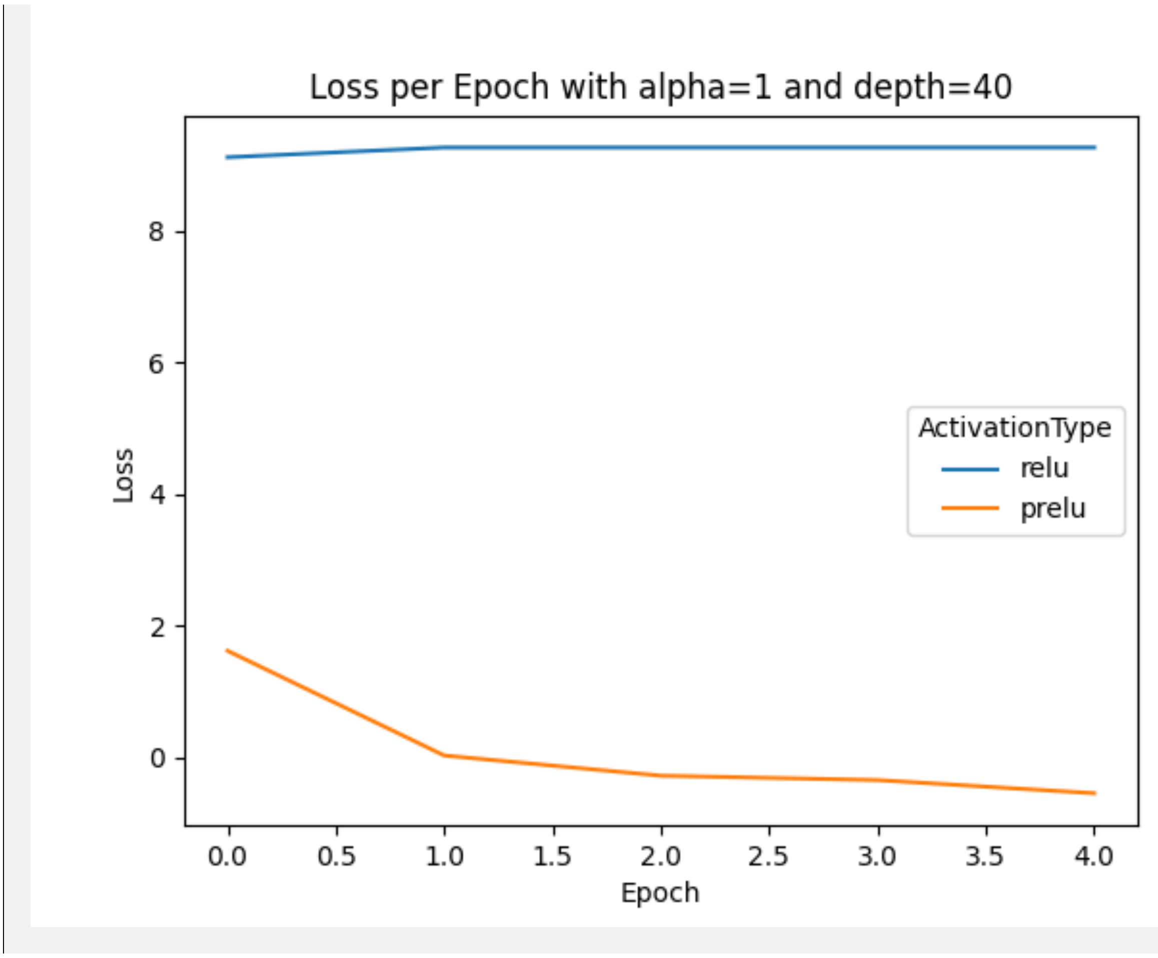

Loss per Epoch with alpha=0.5 and depth=40
Loss
Epoch
ActivationType
relu
prelu
0
2
4
6
8
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

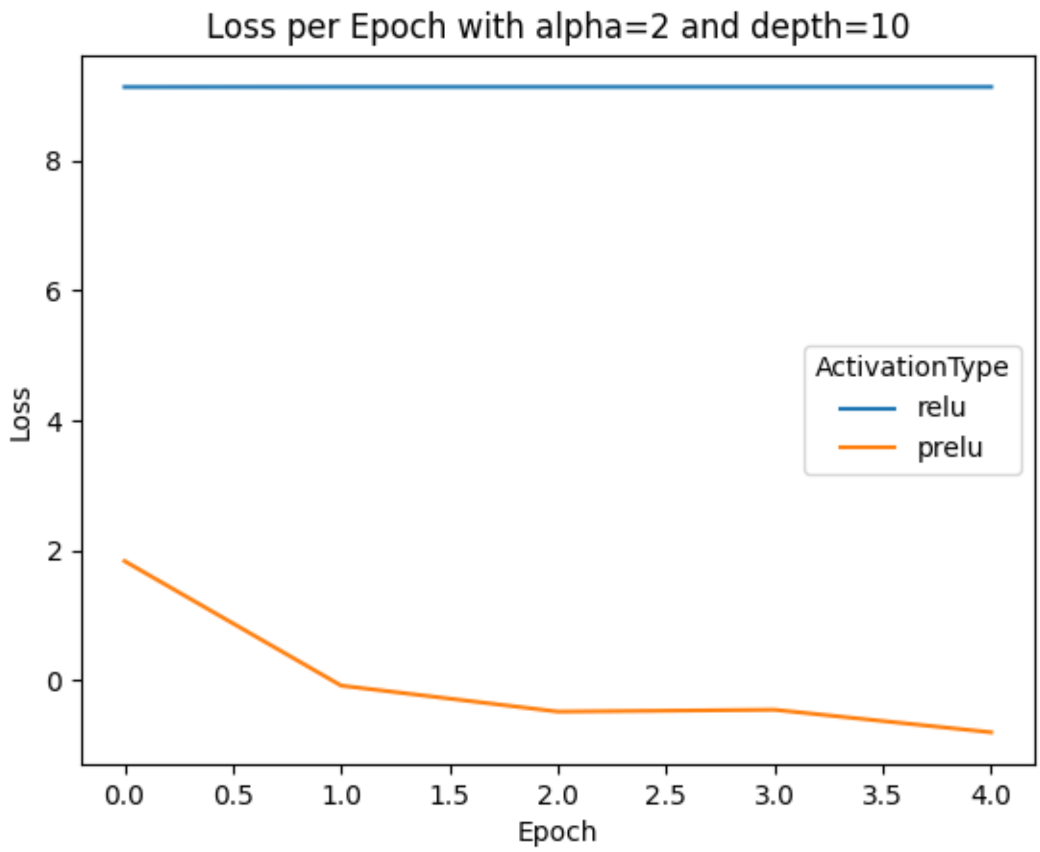

Loss per Epoch with alpha=1 and depth=10
Loss
8
6
4
2
0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Epoch
ActivationType
relu
prelu

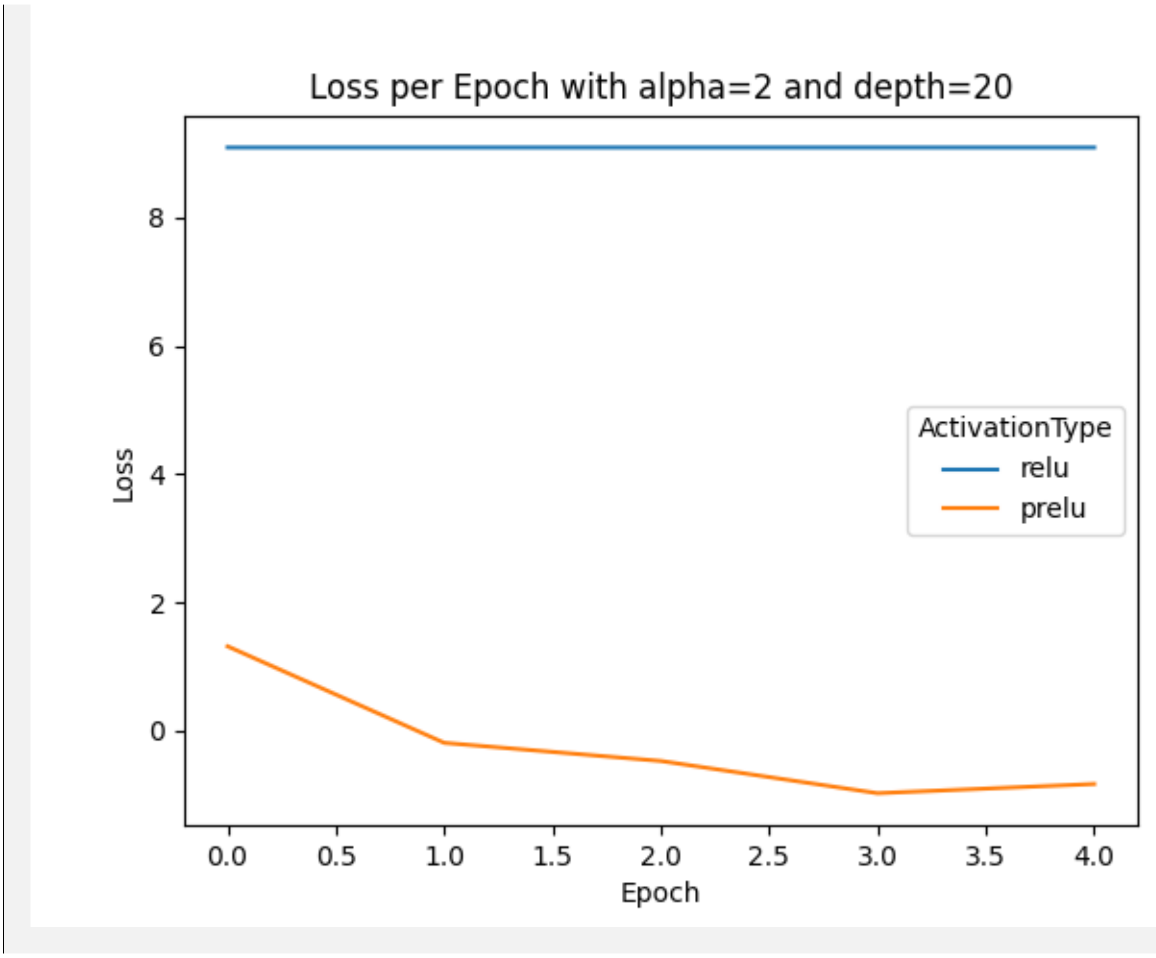


Loss per Epoch with alpha=1 and depth=20
Loss
Epoch
ActivationType
relu
prelu
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0
2
4
6
8

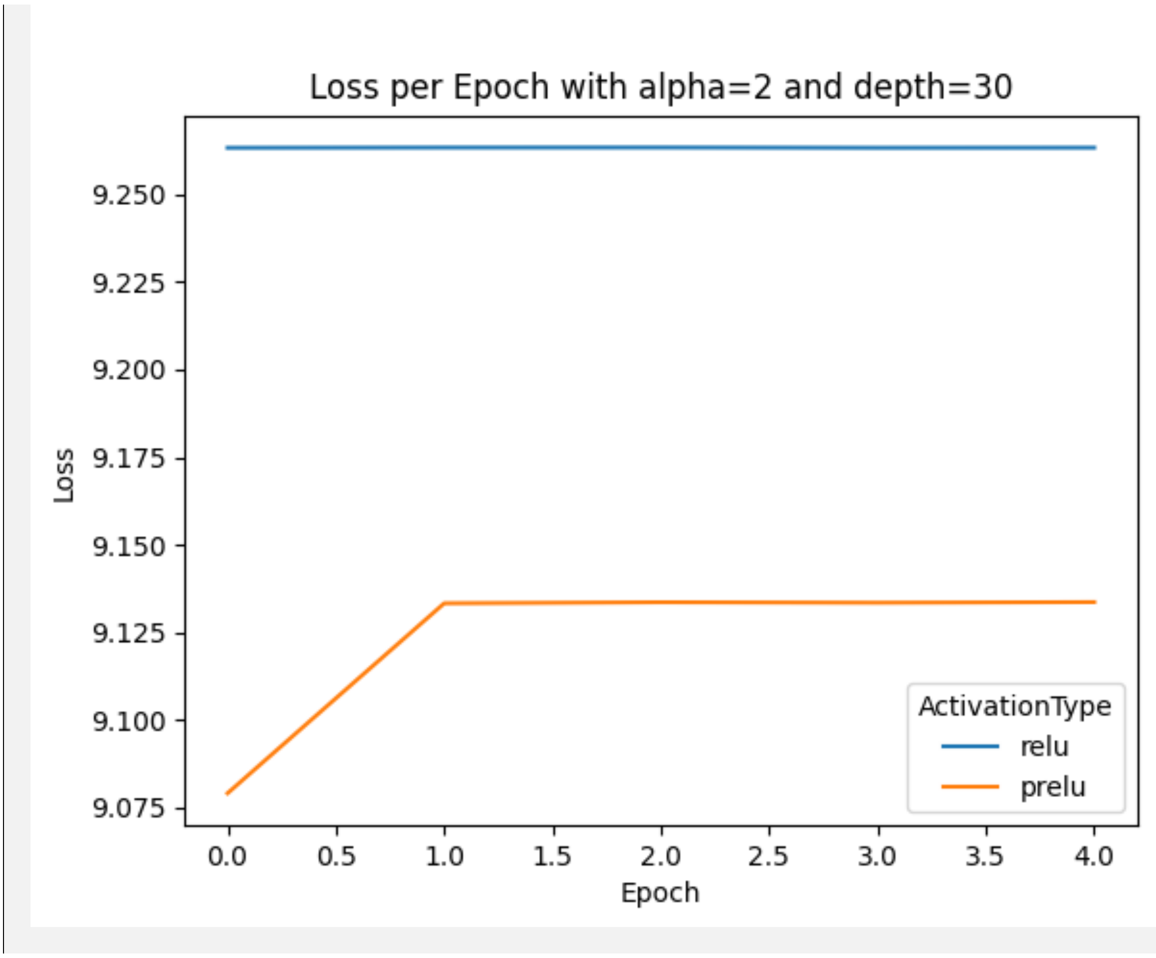

Loss per Epoch with alpha=1 and depth=30
Loss
Epoch
ActivationType
relu
prelu
8
6
4
2
0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

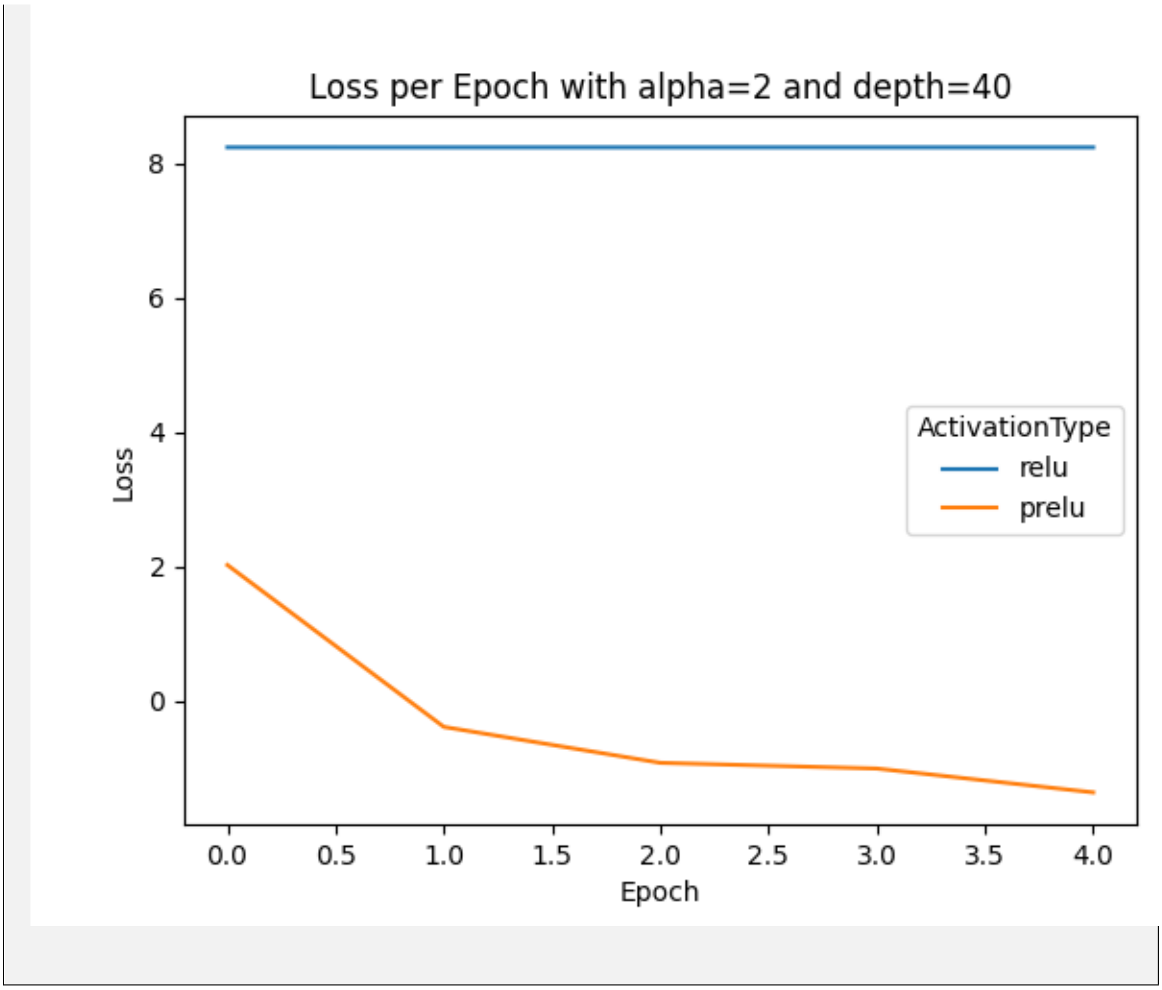

Loss per Epoch with alpha=1 and depth=40
Loss
Epoch
ActivationType
relu
prelu
8
6
4
2
0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0


Loss per Epoch with alpha=2 and depth=10
Loss
Epoch
ActivationType
relu
prelu
0
2
4
6
8
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0



Loss per Epoch with alpha=2 and depth=20
Loss
Epoch
ActivationType
relu
prelu
0
2
4
6
8
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0


Loss per Epoch with alpha=2 and depth=30
Loss
9.250
9.225
9.200
9.175
9.150
9.125
9.100
9.075
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Epoch
ActivationType
relu
prelu


Loss per Epoch with alpha=2 and depth=40
Loss
Epoch
ActivationType
relu
prelu
8
6
4
2
0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

from sklearn.kernel_ridge import KernelRidge

import numpy as np
import pandas as pd
import seaborn as sns

DEBUG_MODE = True
def pp(s):
    if(DEBUG_MODE):
        print(s)

class Net(nn.Module):

    def __init__(self, depth, width, initVariance):
        super(Net, self).__init__()

        self.preLayers = nn.ModuleList()
        for _ in range(depth):
            newLayer = nn.Linear(width, width)
            nn.init.normal_(newLayer.weight, 0, np.sqrt(initVariance))
            self.preLayers.append(newLayer)

        self.outputLayer = nn.Linear(width, 1)

    def forward(self, x):
        pp(f"x dims: {x.shape}")
        for l in self.preLayers:
            x = F.relu(l(x))

        y = F.relu(self.outputLayer(x))
        return y


    def fit(self, xs, ys, learningRate, nEpics=5): # TODO: change back to 10

        optimizer = optim.SGD(self.parameters(), lr=learningRate)
        lossFn = nn.MSELoss()

        lossPerEpoch = []

        for _ in range(nEpics):
            self.train()

            optimizer.zero_grad()
            y_hat = self(xs)
```

```
            pp(f"y_hats : {y_hat}")
            pp(f"ys : {ys}")
            loss_output = lossFn(y_hat, ys)
            loss_output.backward()
            optimizer.step()
            lossPerEpoch.append(loss_output.item()/ys.shape[0])

        return lossPerEpoch

def genData(n, m, d):
    def relu(num):
        return 0 if num < 0 else num
    def genNSphere(r, d):
        v = np.random.normal(0, r, d)
        d = np.sum(v**2) **(0.5)
        return v/d

    xss = [genNSphere(1, d) for _ in range(n)]
    ys = []
    for xs in xss:
        ys.append(sum(map(relu, xs))/m)

    return (np.array(xss), np.array(ys))

def calcError(preds, targets):
    pp(f"predictions: {preds}")
    pp(f"targets: {targets}")
    return ((preds - targets)**2).mean()

def arcKernel(x1: np.array, x2: np.array):
    x1Tx2 = np.dot(x1, x2)
    x1Tx2 = 1.0 if x1Tx2 > 1.0 else x1Tx2
    x1Tx2 = -1.0 if x1Tx2 < -1.0 else x1Tx2
    return x1Tx2*(np.pi - np.arccos(x1Tx2))/(2*np.pi)

def plotResults(df):
    sns.color_palette("Set2")

    #df['Loss'] = np.log(df['Loss'].astype(float))

    title = f"NN vs Kernel Regression"
    plot = sns.lineplot(
        data=df, x="N", y="Loss", hue="ModelType", ci=None
    ).set_title(title)
    plot.figure.savefig(f"{title}.png")
    plot.figure.clf()

def addHistoryRow(df, n, loss, modelType):
    print(df.head())
    new_row = pd.DataFrame(columns=df.columns)
```

```
    new_row.loc[0] = [n, loss, modelType]
    df = pd.concat([df, new_row], ignore_index=True)
    return df

############ BEGIN APPLICATION ############

ns = [20, 40, 80, 160]
d = 10
m = 5
lr = 0.1

historyDf = pd.DataFrame(columns=[
    "N", "Loss", "ModelType"
])

for n in ns:
    trainXs, trainYs = genData(n, m, d)
    testXs, testYs = genData(100, m, d)

    #pp(trainXs)
    #pp(trainYs)
    net = Net(depth=m, width=d, initVariance=1/m)
    lossPerEpoch = net.fit(torch.Tensor(trainXs), torch.Tensor(trainYs), lr)
    pp(f"Loss per eopch: {lossPerEpoch}")
    nnError = calcError(net(torch.Tensor(testXs)).detach().numpy().flatten(), testYs)

    historyDf = addHistoryRow(historyDf, n, nnError, "Neureal Network")

    krr = KernelRidge(kernel=arcKernel)
    print(trainXs)
    krr.fit(trainXs, trainYs)
    krrError = calcError(krr.predict(testXs), testYs)
    historyDf = addHistoryRow(historyDf, n, krrError, "Kernel Regression")


plotResults(historyDf)
```

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader, Subset

import torchvision
from torchvision.datasets import MNIST
from torchvision import transforms

import numpy as np
import pandas as pd
import seaborn as sns

DEBUG_MODE = True
def pp(s):
    if(DEBUG_MODE):
        print(s)

class Net(nn.Module):

    def __init__(self, depth, alpha, width, initVariance):
        super(Net, self).__init__()
        self.alpha = alpha

        #self.conv1 = nn.Conv2d(1, 6, 5)
        #self.conv2 = nn.Conv2d(6, 16, 5)

        self.preLayers = nn.ModuleList()
        for _ in range(depth):
            newLayer = nn.Linear(width, width)
            #pp(f"initVariance: {initVariance}")
            nn.init.normal_(newLayer.weight, 0, np.sqrt(initVariance))
            self.preLayers.append(newLayer)

        self.outputLayer = nn.Linear(width, 1)
        self.outSig = nn.Sigmoid()

    def forward(self, x):

        x = torch.flatten(x, 2, 3)

        for l in self.preLayers:
            x = F.leaky_relu(l(x), self.alpha)

        x = F.leaky_relu(self.outputLayer(x), self.alpha)

        return torch.squeeze(self.outSig(x))


```

```
    def fit(self, batches, learningRate, epochs_n=5): # TODO: change back to 10
        loss_per_epoch = []

        optimizer = optim.SGD(self.parameters(), lr=learningRate)
        lossFn = nn.BCELoss()

        for i in range(epochs_n):
            batch_loss = 0

            self.train()
            for image, label in batches:
                optimizer.zero_grad()
                y_hat = self(image)
                #pp(f"y_hat: {y_hat}" )
                #pp(f"label: {label}" )
                loss = lossFn(y_hat, label.float())
                loss.backward()
                optimizer.step()
                #pp(f"loss: {loss}")
                batch_loss += loss.item()

            loss_per_epoch.append(batch_loss)

        return loss_per_epoch

# setup data
trainDataRaw = MNIST(
    root='data',
    train=True,
    transform=transforms.Compose([
        transforms.Resize(16),
        transforms.ToTensor(),
    ]),
    download=True
)


def addToDf(df: pd.DataFrame, history, actType):
    for i, loss in enumerate(history):
        new_row = pd.DataFrame(columns=df.columns)
        new_row.loc[0] = [i, loss, actType]
        df = pd.concat([df, new_row], ignore_index=True)

    return df

def plotResults(df, d, a, lr):
    sns.color_palette("Set2")

    df['Loss'] = np.log(df['Loss'].astype(float))

```

```
    title = f"Loss per Epoch with alpha={a} and depth={d}"
    plot = sns.lineplot(
        data=df, x="Epoch", y="Loss", hue="ActivationType", ci=None
    ).set_title(title)
    plot.figure.savefig(f"{title}_{d}_{a}.png")
    plot.figure.clf()

########### DO THE STUFF ###########
batchSize = 64
width = 256
depths = [10, 20, 30, 40]
alphas = [2, 1, 0.5, 0.1]

keepIdxs = (trainDataRaw.targets==0) | (trainDataRaw.targets==1)
trainDataRaw.targets = trainDataRaw.targets[keepIdxs]
trainDataRaw.data = trainDataRaw.data[keepIdxs]

trainBatches = DataLoader(trainDataRaw, batch_size=batchSize, shuffle=True)

initFuncs = {
    "relu": (lambda alpha: 2/width),
    "prelu": (lambda alpha: 2/(width*(1 + alpha**2))),
}

history = []
learningRate = 0.01

for depth in depths:
    for alpha in alphas:

        historyDf = pd.DataFrame(columns=[
            "Epoch", "Loss", "ActivationType"
        ])

        for activationType, initFunc in initFuncs.items():
            net = Net(depth, alpha, width, initFunc(alpha))
            losses = net.fit(trainBatches, learningRate, epochs_n=5)
            pp(f"Losses: {losses}")
            historyDf = addToDf(historyDf, losses, activationType) # TODO

        print(historyDf.head())
        plotResults(historyDf, d=depth, a=alpha, lr=learningRate)
```