YANIS TAZI HOMEWORK. 2 DEEP LEARNING SYSTEMS

yt1600@nyu.edu

In [2]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.colors import ListedColormap
import math
import random
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.ensemble import GradientBoostingClassifier
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.filterwarnings('ignore')
import time
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix
from sklearn.linear_model import Perceptron
from sklearn.linear_model import Ridge
from sklearn import linear_model
from sklearn.ensemble import AdaBoostClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import f1_score
from sklearn.metrics import auc
import copy
import seaborn as sns
import tensorflow as tf
```

Problem 2

Q1)

In [3]:
```
import keras
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from keras import initializers
from keras.datasets import mnist
import tensorflow as tf
from utils import (

    get_init_id,
    grid_axes_it,
    compile_model,
    create_cnn_model,
    LossHistory,
    compile_model,
    create_mlp_model,
    get_activations,
    grid_axes_it,
)
```

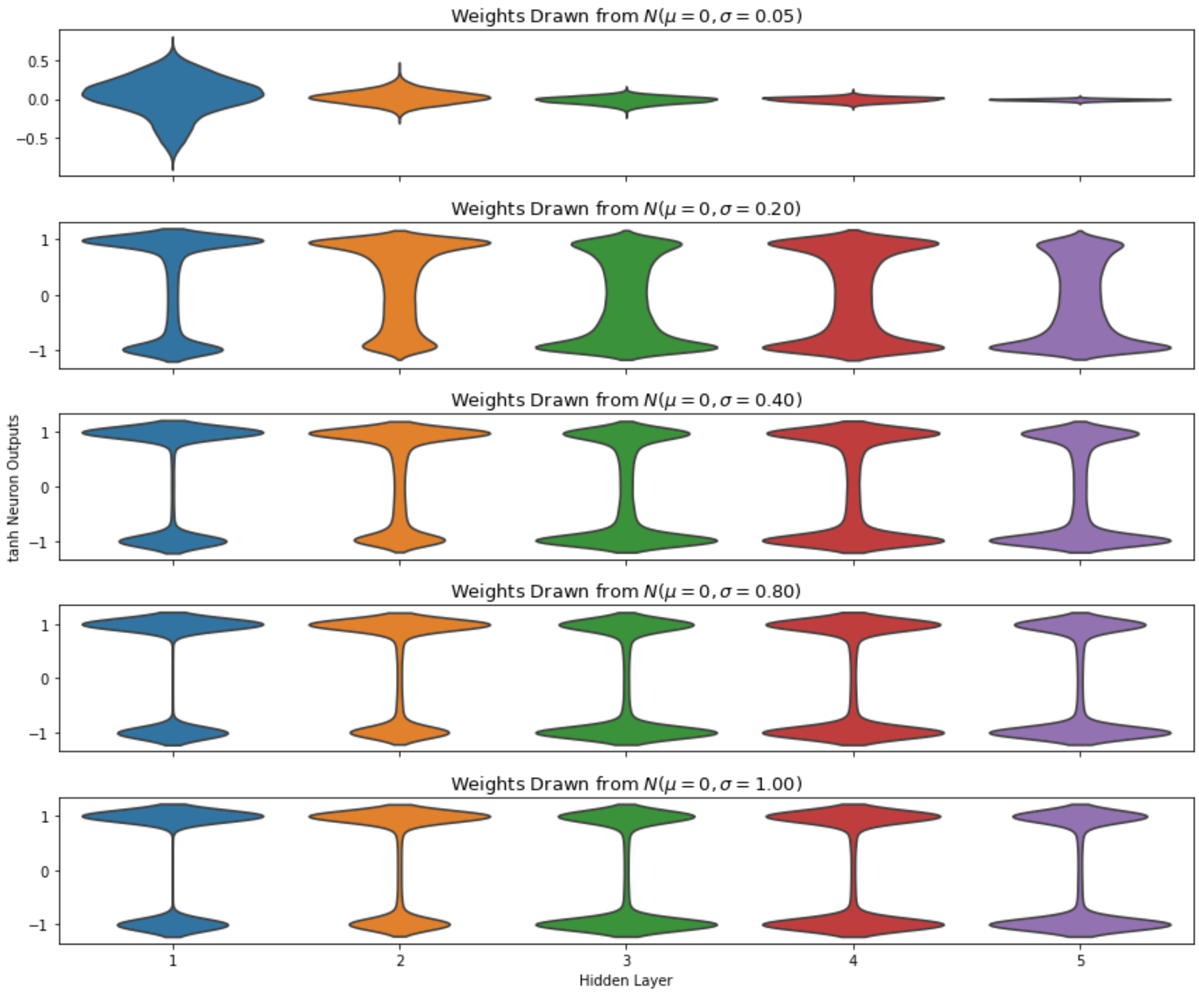
In [4]:
```
def plot_output(activation='relu',sigmas=[0.10, 0.14, 0.28]):



    seed = 10

    # Number of points to plot
    n_train = 1000
    n_test = 100
    n_classes = 10

    # Network params
    n_hidden_layers = 5
    dim_layer = 100
    batch_size = n_train
    epochs = 1

    # Load and prepare MNIST dataset.
    n_train = 60000
    n_test = 10000

    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    num_classes = len(np.unique(y_test))
    data_dim = 28 * 28

    x_train = x_train.reshape(60000, 784).astype('float32')[:n_train]
    x_test = x_test.reshape(10000, 784).astype('float32')[:n_train]
    x_train /= 255
    x_test /= 255

    y_train = keras.utils.to_categorical(y_train, num_classes)
    y_test = keras.utils.to_categorical(y_test, num_classes)

    # Run the data through a few MLP models and save the activations fro
m
    # each layer into a Pandas DataFrame.
    rows = []
    sigmas = sigmas
    for stddev in sigmas:
        init = initializers.RandomNormal(mean=0.0, stddev=stddev, seed=s
eed)
        activation = activation

        model = create_mlp_model(
            n_hidden_layers,
            dim_layer,
            (data_dim,),
            n_classes,
            init,
            'zeros',
            activation
        )
        compile_model(model)
        output_elts = get_activations(model, x_test)
        n_layers = len(model.layers)
        i_output_layer = n_layers - 1
```

```
        for i, out in enumerate(output_elts[:-1]):
            if i > 0 and i != i_output_layer:
                for out_i in out.ravel()[::20]:
                    rows.append([i, stddev, out_i])

    df = pd.DataFrame(rows, columns=['Hidden Layer', 'Standard Deviatio
n', 'Output'])

    # Plot previously saved activations from the 5 hidden layers
    # using different initialization schemes.
    fig = plt.figure(figsize=(12,2*len(sigmas)))
    axes = grid_axes_it(len(sigmas), 1, fig=fig)
    for sig in sigmas:
        ax = next(axes)
        ddf = df[df['Standard Deviation'] == sig]
        sns.violinplot(x='Hidden Layer', y='Output', data=ddf, ax=ax, sc
ale='count', inner=None)

        ax.set_xlabel('')
        ax.set_ylabel('')

        ax.set_title('Weights Drawn from $N(\mu = 0, \sigma = {%.2f})$'
% sig, fontsize=13)

        if sig == sigmas[int(len(sigmas)/2)]:
            ax.set_ylabel(activation +" Neuron Outputs")
        if sig != sigmas[-1]:
            ax.set_xticklabels(())
        else:
            ax.set_xlabel("Hidden Layer")

    plt.tight_layout()
    plt.show()
```

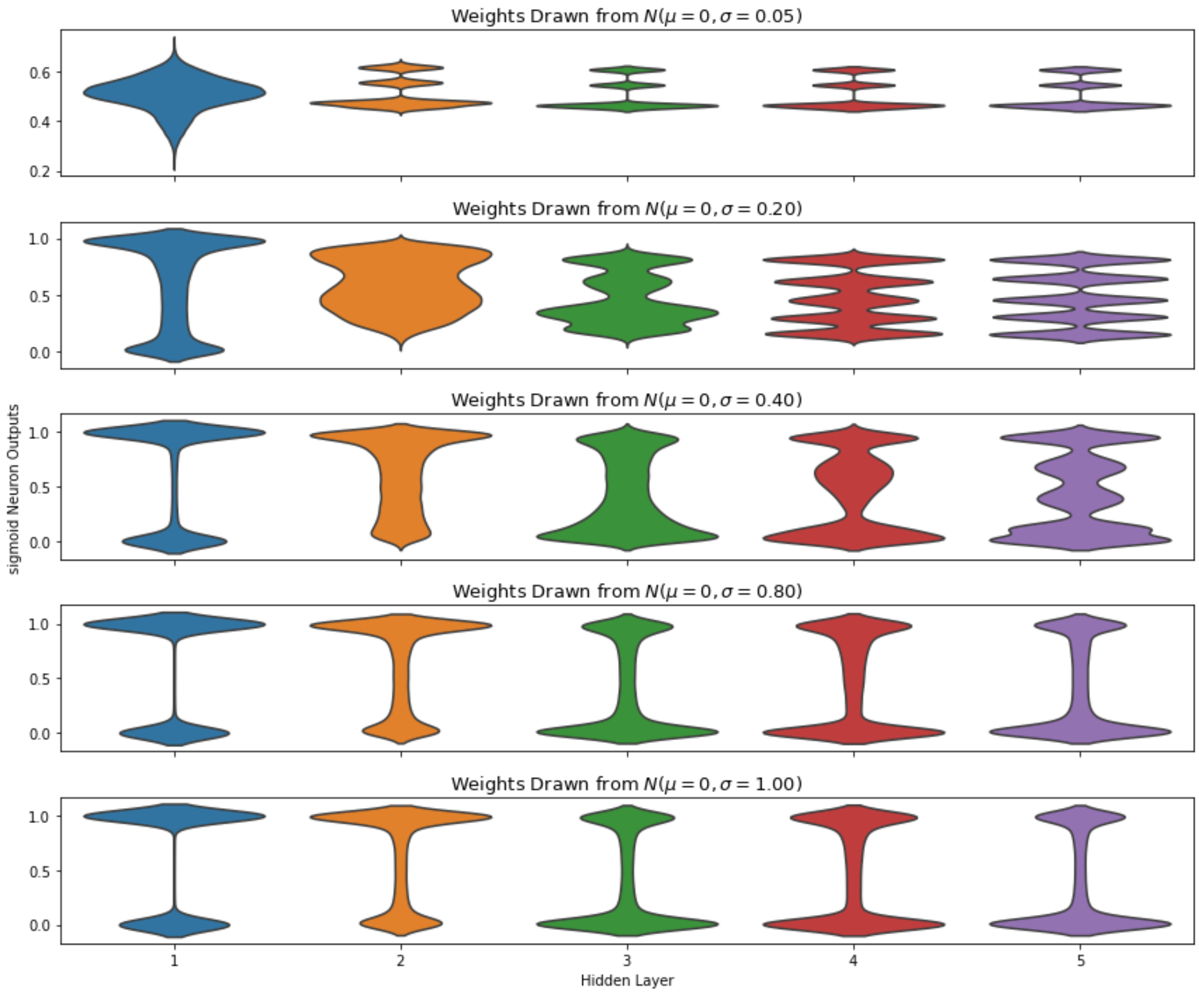
Sigmoid and tanh

In [5]:
```
print('tanh')
plot_output(activation='tanh',sigmas=[0.05,0.2,0.4,0.8,1])
print('sigmoid')
plot_output(activation='sigmoid',sigmas=[0.05,0.2,0.4,0.8,1])
```

tanh



sigmoid



Vanishing and exploding gradient can be explained by understanding the relationship between gradient and activation outpus :

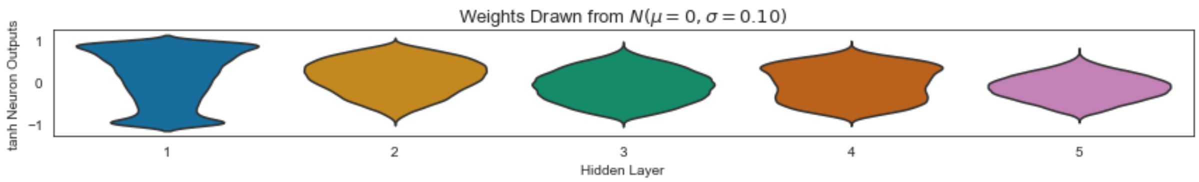
Indeed, $\frac{dL}{dW^i_{jk}} = \frac{dL}{dx^{i+1}_k}\frac{dx^{i+1}_k}{dW^i_{jk}}$**, with :** $\frac{dx^{i+1}_k}{dW^i_{jk}} = f'(s^i_j)x^i_j$**.**

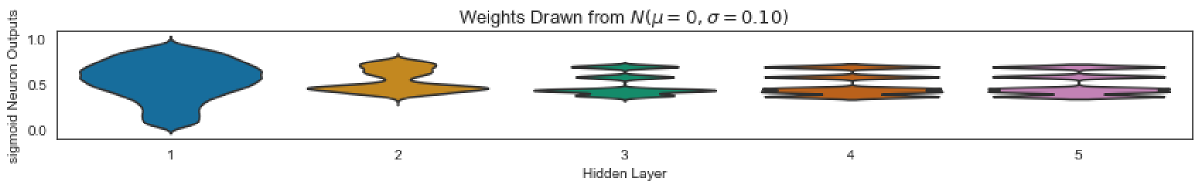
Therefore, if the output goes to 0 the gradient as well. Similarly, the derivatives of the activations f goes to 0 for extreme values of s_j. Hence the choice of the activation is also primordial. We see that for small standard deviation (0.05) and tanh activation , we have that the gradient is vanishing.

Sigmoid and tanh with Glorot initialization

Glorot initialization : $\sigma = \sqrt{\frac{1}{n_i}}$**, when we sample from normal distribution with 0 mean and** $n_i = n_{i+1}$**.**

In [46]:
```
plot_output(activation='tanh',sigmas=[np.sqrt(2/(100+100))])
plot_output(activation='sigmoid',sigmas=[np.sqrt(2/(100+100))])
```



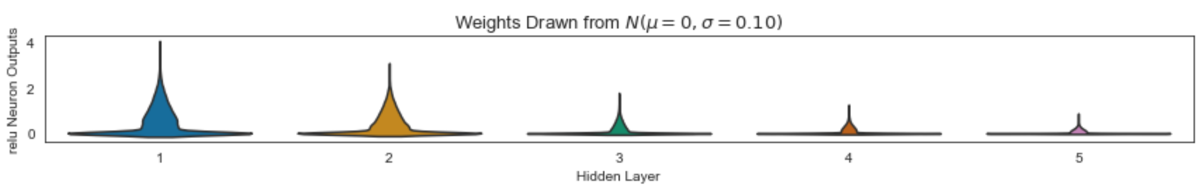


Glorot works well here

Relu

Glorot initialization : $\sigma = \sqrt{\frac{1}{n_i}}$, when we sample from normal distribution with 0 mean and $n_i = n_{i+1}$.

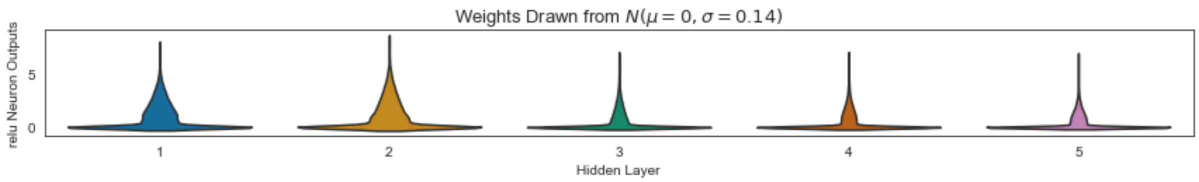
In [51]:
```
plot_output(activation='relu',sigmas=[np.sqrt(1/(100))])
```



It is not working !

He initialization : $\sigma = \sqrt{\frac{2}{n_i}}$, when we sample from normal distribution with 0 mean

```
In [50]: plot_output(activation='relu',sigmas=[np.sqrt(2/(100))])
```



It works and makes sense intuitively since Relu outputs zeroes for all negative values corresponding to reducing the variance by half . Therefore, multiplying σ by $\sqrt{2}$ overcome this problem !

Q2)

The authors proposed a new initialization procedure to overcome dying Relu called RAI and focuses on the worst case of dying Relu where the entire network dies and therefore the network is just a constant.

In [14]:
```
def run_sequential_model(num_simulations=1000,num_examples=3000,activati
on=tf.nn.relu,batch_size=64,epochs=10):
    collapse = []
    for i in range(num_simulations):
        if(i%50==0):
            print('Simulation number :' + str(i))
        np.random.seed(i)
        x_train = np.random.uniform(-np.sqrt(7),np.sqrt(7),size=num_exam
ples)
        y_train = [x * np.sin(5*x) for x in x_train]
        tf.random.set_seed(i)
        model = keras.Sequential([
            keras.layers.Dense(2, activation=activation),
            keras.layers.Dense(2, activation=activation),
            keras.layers.Dense(2, activation=activation),
            keras.layers.Dense(2, activation=activation),
            keras.layers.Dense(2, activation=activation),
            keras.layers.Dense(2, activation=activation),
            keras.layers.Dense(2, activation=activation),
            keras.layers.Dense(2, activation=activation),
            keras.layers.Dense(2, activation=activation),
            keras.layers.Dense(2, activation=activation),
            keras.layers.Dense(1)
          ])

        optimizer = tf.keras.optimizers.RMSprop(0.0099)
        model.compile(loss='mean_squared_error',optimizer=optimizer)
        model.fit(x_train,np.array(y_train),batch_size=batch_size,epochs
=epochs,verbose=0)
        collapse += [len(np.unique((model.predict(np.linspace(1,10,10
)))))]
    return collapse
```

In [ ]:
```
collapse_relu = run_sequential_model(num_simulations=1000,activation=tf.
nn.relu)
```

In [9]:
```
num_simulations  = 1000
print('Percentage that collapse for Relu :')
100*collapse_relu.count(1)/num_simulations
```

```
Percentage that collapse for Relu :
```

Out[9]: 96.7

Q 3)

In [ ]:
```
collapse_leaky_relu = run_sequential_model(num_simulations=1000,activati
on=tf.nn.leaky_relu)
```

In [17]:
```
num_simulations  = 1000
print('Percentage that collapse for Leaky Relu :')
100*collapse_leaky_relu.count(1)/num_simulations
```

```
Percentage that collapse for Leaky Relu :
```

Out[17]:
```
0.1
```

Yes, leaky relu significantly reduced the neural netowrk collapse !

In [ ]: