# Not So Linear Regression (50 Total Points)

Often times linear regression is presented in terms of input features, which results in linear models (shocker!). Linear models are an already powerful technique, but they are super-charged when constructed atop of *non*linear features. Arguebly, the constrution of linear models with nonlinear features is one of the strongest (and most used) tricks in ML and is core to neural networks. Below we explore how the expressiveness of linear models increases based on features. This highlights the importance of the question, "*Linear in what?*"

## Data and Setup

First, it may help to enable GPUs for the notebook:

- Navigate to Edit→Notebook Settings
- select GPU from the Hardware Accelerator drop-down

Next, confirm that we can connect to the GPU with tensorflow.

*(Note, it is fine if you can not connect to GPU, it just might take a little longer to run.)*

```
%tensorflow_version 2.x
import tensorflow as tf
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))
```

```
    Found GPU at: /device:GPU:0
```

Let's import additional packages of use.

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

**Generate data.**

Generate $N$ data samples of input (1d) features $X$ and outputs $Y$, which shall depend non-linearly on $X$.

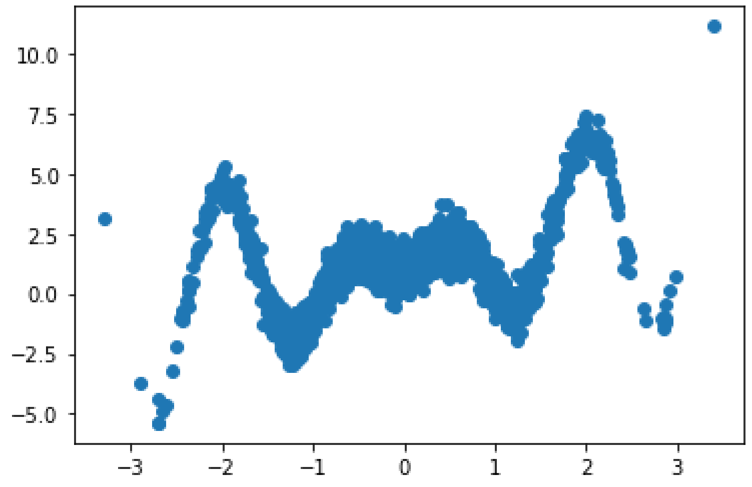
```
d = 1
N = 2000
```

```
X = tf.random.normal((N, 1))

def f(x):
  return tf.exp(0.5*x)+2.0*x*tf.sin(4.0*x)

sigma = 0.5
Y = f(X) + sigma*tf.random.normal((N,1))
```

Visualize the data.

```
plt.scatter(X, Y)
```

```
    <matplotlib.collections.PathCollection at 0x7f32403bde10>
```



## Linear Regression (17 Points)

**(12 Points)** First let's implement a function that returns the linear coefficients for ordinary least squares regression.

```
def linear_coefs(X, Y):
  """
  Args:
    X: N x d matrix of input features
    Y: N x 1 matrix (column vector) of output response

  Returns:
    Beta: d x 1 matrix of linear coefficients
  """
  prepend_ones = tf.ones([N, 1], dtype=tf.dtypes.float32)
  X_prep = tf.concat([prepend_ones, X], axis=1)

  xTx = tf.linalg.matmul(tf.transpose(X_prep), X_prep)
  xTy = tf.linalg.matmul(tf.transpose(X_prep), Y)
  print(xTx.shape, xTy.shape)
  beta = tf.linalg.solve(xTx, xTy)
  print(beta)

  # TODO: here you are restricted (have to) use tf.linalg.solve for the assignment

  return beta[0], beta[1:]
```

**(5 Points)** Clearly, the groundtruth relationship is not a linear one. However, let's see what the best linear fit is for this data.

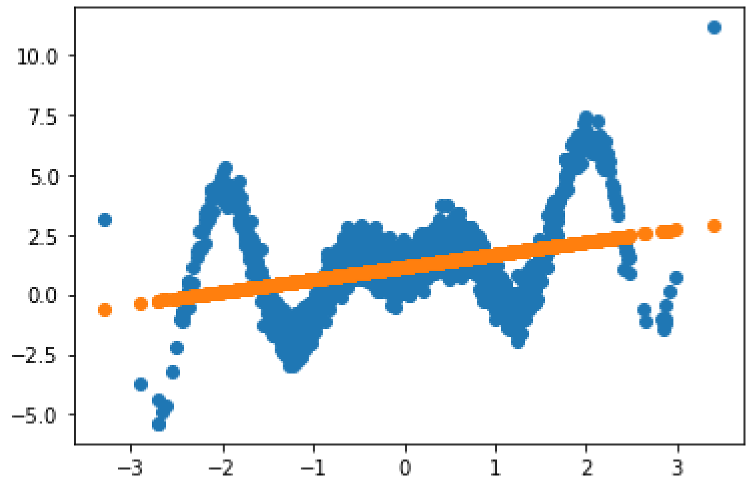
```
# TODO: be sure to account for an intercept term
# beta_linear = linear_coefs(...)
beta_intercept, beta_linear = linear_coefs(X, Y)
Y_linear = tf.linalg.matmul(X, beta_linear) + beta_intercept  # should be the predicte
```

```
    (2, 2) (2, 1)
    tf.Tensor(
    [[1.1499931]
     [0.523499 ]], shape=(2, 1), dtype=float32)
```

Let's plot the linear predictions.

```
plt.scatter(X, Y)
plt.scatter(X, Y_linear)
```

```
    <matplotlib.collections.PathCollection at 0x7f3240330c10>
```



As you can see this captures some trend, but leaves a lot to be desired.

## Polynomial Regression (16 Points)

Let's turn back time and revist an algebra course classic: polynomials! Recall that polynomial are made up of monomials, which are products of variable (features) take to powers. With 1d data, these are terms like $x^2$, $x^7$, etc. When we fit a polynomial, we are looking for $\beta$ coeficients to $\hat{f}(x) = \sum_{j=1}^{m} \beta_j x^j + \beta_0$. Does this look familiar? (It should!)

**(16 points)** Let's fit a polynomial using OLS. In particular, let's fit a polynomial of order $8$.

```
from sklearn.preprocessing import PolynomialFeatures

# First generate the features corresponding to a 8th order 1d polynomial.
# You'll lose points :( if you do not do it in a simple, single line
X_poly = PolynomialFeatures(degree=8).fit_transform(X)[:, 1:]
print(X_poly.shape)

# Lets get the corresponding polynomial coefficients
print(X_poly.shape, Y.shape)
intercept, beta_poly = linear_coefs(X_poly, Y)

# Finally lets plot the estimates with the polynomial
Y_poly = np.matmul(X_poly, beta_poly) + intercept

plt.scatter(X, Y)
plt.scatter(X, Y_poly)
```

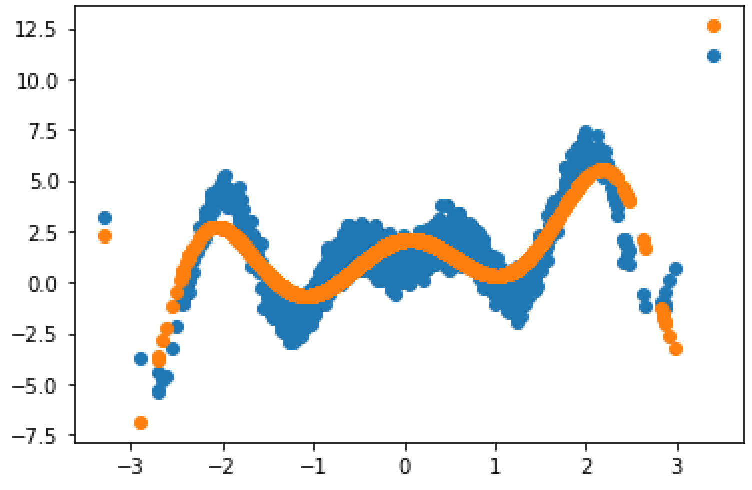
```
    (2000, 8)
    (2000, 8) (2000, 1)
    (9, 9) (9, 1)
```

Let's plot the polynomial estimates.

```
     [ 5.80598020e-01]
```

```
plt.scatter(X, Y)
plt.scatter(X, Y_poly)
```

```
    <matplotlib.collections.PathCollection at 0x7f324017cd10>
```



A much better fit! In the future we'll discuss how to choose hyperparemeters to our model (such as the order of polynomials). For now, just play around with the order to see how estimates vary.

## Random-Feature Regression (10 Points)

Lastly, we'll consider a seemingly crazy idea: performing linear regression over *random* features. That is, we shall fit a model $\hat{f}(x) = \sum_{j=1}^{m} \beta_j \phi_j(x)$. Where $\phi_j(x)$ are randomly constructed features. In particular, we shall construct $\phi_j(x) = \cos(\omega_j^T x + b_j)$ where $\omega_j$ was drawn from a Gaussian, and $b_j$ from a Uniform distribution. Note that $(\omega_j, b_j)$s are draw randomly once, and are then *held fixed* for the remainer of their usage. (Think about what goes wrong if they are redrawn randomly each time.) Although this seems adhoc, a linear model on such random features are actually approximating a very flexible class of functions (see https://people.eecs.berkeley.edu/~brecht/papers/07.rah.rec.nips.pdf).

```
def rand_feats(X, D=1000, gamma=1.0):
  """
  Args:
    X: N x d matrix of input features
    D: Integer, number of random features
    gamma: Float, scale of frequencies
  Returns:
    Phis: N x D matrix of random features
  """
  d = X.get_shape()[1]
  tf.random.set_seed(112358)  # Why is this needed?
  Ws = gamma*tf.random.normal((d, D))
  #print(Ws)
  bs = 2.0*np.pi*tf.random.uniform((1, D))
  #print(bs)
  XWs = tf.matmul(X, Ws)
  return tf.cos(XWs+bs)
```

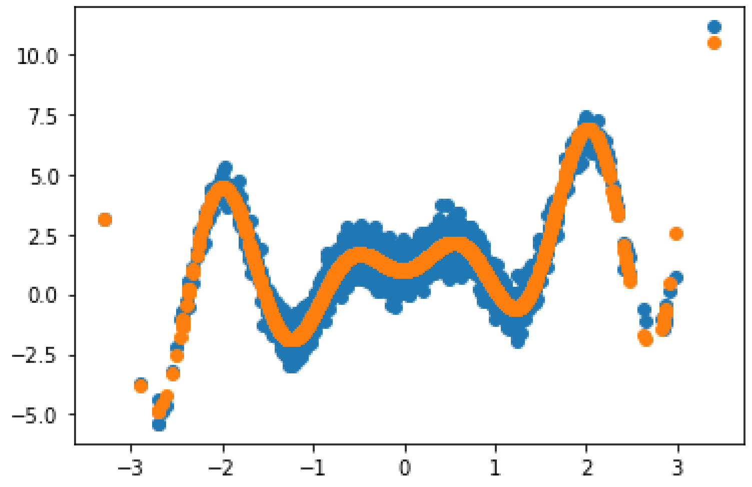
**(7 Points)** Get predictions based on the random features above.

```
# TODO: Get the Y estimates using random features. (Default D, gamma is fine.)
X_rands = rand_feats(X)
print(X_rands.shape)

intercept, beta_rands = linear_coefs(X_rands, Y)
Y_randfeats = np.matmul(X_rands, beta_rands) + intercept
```

```
(2000, 1000)
(1001, 1001) (1001, 1)
tf.Tensor(
[[-32.82667  ]
 [ 28.77658  ]
 [-42.440662 ]
 ...
 [-42.460423 ]
 [-10.120999 ]
 [ -4.6384945]], shape=(1001, 1), dtype=float32)
```

```
plt.scatter(X, Y)
plt.scatter(X, Y_randfeats)
```

```
<matplotlib.collections.PathCollection at 0x7f32400fe9d0>
```



Not bad for a bunch of random features! This, along with kernels, are pretty powerful concepts. I highly recommend looking further into random features.

**(10 Points)** Why is it necessary to set the random seed `tf.random.set_seed(112358)` in the function `rand_feats` above. (Think about what would go wrong if we do not do this.)

1. Reproducibility
2. We want to use the same set of features (even though they were picked randomly in the beginning)

✓ 0s completed at 11:55 PM ● ✕