Causal Inference - HW3

Maxime Tchibozo (MT3390)

$U_1 \sim \mathbf{N}(0, 1)$

$U_2 \sim \mathbf{N}(\beta_{U_1,U_2} U_1, 1)$

$X_1 \sim Binomial(1, p = logit^{-1}(\beta_{U_1,X_1} U_1))$

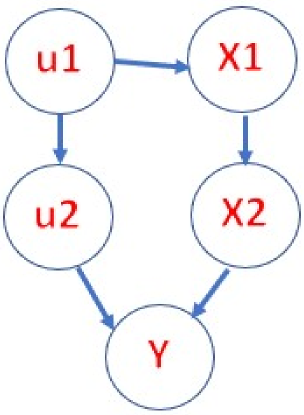
$X_2 \sim \mathbf{N}(\beta_{X_1,X_2} X_1, 1)$

$Y \sim \mathbf{N}(\beta_{X_2,Y} X_2 + \beta_{U_2,Y} U_2, 1)$

(a) Draw the causal graph for this system

In [0]:
```
from IPython.display import Image
Image("graph.png")
```

Out[0]:



(b) Choose a set of back-door variables for the effect of X_1 on Y

The set Z={U_1} is a satisfactory backdoor set:

(i) it does not contain descendents of X_1

(ii) it blocks all paths from X_1 to Y with an arrow into X_1.

(c) Code this data-generating process in Python choosing parameters β that result in at least 50% bias for the causal effect of X_1 on Y.

(d) What are the true and naïve average treatment effects (ATEs), δ, in your data-generating process?

In [4]:
```
from numpy.random import normal
import numpy as np
import pandas as pd

N=10000

beta_U1_U2 = 1
beta_U1_X1 = 2
beta_X1_X2 = 3
beta_X2_Y = 4
beta_U2_Y = 5

U1 = normal(size=N)
U2 = normal(beta_U1_U2*U1)
z = 1/(1 + np.exp(-beta_U1_X1*U1))
X1 = np.random.binomial(n=1,p=z)
X2 = normal(beta_X1_X2*X1)
Y = normal(beta_X2_Y*X2+beta_U2_Y*U2)

df = pd.DataFrame({'U1':U1,'U2':U2,'X1':X1,'X2':X2,'Y':Y})

delta_naive = df[df.X1==1]['Y'].mean()-df[df.X1==0]['Y'].mean()
delta_naive_error_percentage = (delta_naive-beta_X1_X2*beta_X2_Y)/(beta_X1_X2*be
ta_X2_Y)*100

print('The naive caual effect is : ',delta_naive)
print('The bias of the naive estimate is : {}%'.format(delta_naive_error_percent
age))
```

```
The naive caual effect is :  18.171968643429786
The bias of the naive estimate is : 51.43307202858155%
```

The true naive estimate is $\beta_{X_2,Y} \cdot \beta_{X_1,X_2} = 12$

(e) Use OLS regression with your back-door set to make an unbiased estimate of the ATE. What is your estimate and 95% confidence interval?

In [5]:
```
from statsmodels.regression.linear_model import OLS

df['intercept'] = 1
model = OLS(endog=df['Y'],exog=df[['X1','U1','intercept']])
results = model.fit()
results.summary()
```

Out[5]:

OLS Regression Results

Dep. Variable:	Y	**R-squared:**	0.701
Model:	OLS	**Adj. R-squared:**	0.701
Method:	Least Squares	**F-statistic:**	1.171e+04
Date:	Sat, 07 Mar 2020	**Prob (F-statistic):**	0.00
Time:	03:35:38	**Log-Likelihood:**	-32847.
No. Observations:	10000	**AIC:**	6.570e+04
Df Residuals:	9997	**BIC:**	6.572e+04
Df Model:	2		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
X1	12.1817	0.163	74.819	0.000	11.863	12.501
U1	4.9583	0.082	60.490	0.000	4.798	5.119
intercept	-0.1558	0.104	-1.494	0.135	-0.360	0.049

Omnibus:	2.082	**Durbin-Watson:**	2.023
Prob(Omnibus):	0.353	**Jarque-Bera (JB):**	2.058
Skew:	-0.014	**Prob(JB):**	0.357
Kurtosis:	2.935	**Cond. No.**	3.50

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Using OLS on the backdoor set gives an estimate of the ATE of 12.1817, which is around 1% error. The 95% confidence interval is $[11.863, 12.501]$

(f) Use inverse propensity weighted OLS regression with no covariates to make an unbiased estimate of the ATE. What is your estimate and the 95% confidence interval?

We will estimate the propensity scores using a Logit function.

In [6]:
```
from statsmodels.discrete.discrete_model import Logit
model = Logit(endog=df['X1'],exog=df[['U1','intercept']])
result = model.fit()
df['propensity_score'] = df['X1']*result.predict()+(1-df['X1'])*(1-result.predic
t())
```

```
Optimization terminated successfully.
         Current function value: 0.459925
         Iterations 7
```

In [7]:
```
from statsmodels.api import WLS

df['weight'] = 1/df['propensity_score']
model = WLS(endog=df['Y'],exog=df[['intercept','X1']],weights=df['weight'])
result = model.fit()
result.summary()
```

Out[7]:

WLS Regression Results

Dep. Variable:	Y	**R-squared:**	0.373
Model:	WLS	**Adj. R-squared:**	0.373
Method:	Least Squares	**F-statistic:**	5937.
Date:	Sat, 07 Mar 2020	**Prob (F-statistic):**	0.00
Time:	03:36:13	**Log-Likelihood:**	-36227.
No. Observations:	10000	**AIC:**	7.246e+04
Df Residuals:	9998	**BIC:**	7.247e+04
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
intercept	-0.2472	0.114	-2.162	0.031	-0.471	-0.023
X1	12.5138	0.162	77.052	0.000	12.195	12.832

Omnibus:	2043.971	**Durbin-Watson:**	2.023
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	75141.555
Skew:	0.038	**Prob(JB):**	0.00
Kurtosis:	16.429	**Cond. No.**	2.61

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Inverse propensity weighted OLS regression with no covariates gives an estimate of 12.5138 and a 95% confidence interval of $[12.195, 12.832]$

(g) Use a higher capacity machine learning model to estimate the ATE using the technique from section 5.3.3. What is your estimate and the 95% confidence interval?

In [8]:
```
from keras.layers import Dense, Input
from keras.models import Model

x_in = Input(shape=(1,))
h1 = Dense(128,activation='tanh')(x_in)
h2 = Dense(128,activation='tanh')(h1)
h3 = Dense(128,activation='tanh')(h2)
h4 = Dense(128,activation='tanh')(h3)
y = Dense(1,activation='linear')(h4)

model = Model(inputs=[x_in],outputs=[y])
model.compile('adam',loss='mse',metrics=['mse'])
model.fit(df[['X1']],df[['Y']], batch_size=1024,epochs=20,sample_weight=df['weig
ht'])
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/t
ensorflow_backend.py:1033: The name tf.assign_add is deprecated. Please use t
f.compat.v1.assign_add instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/t
ensorflow_backend.py:1020: The name tf.assign is deprecated. Please use tf.com
pat.v1.assign instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/t
ensorflow_backend.py:3005: The name tf.Session is deprecated. Please use tf.co
mpat.v1.Session instead.

Epoch 1/20
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/t
ensorflow_backend.py:190: The name tf.get_default_session is deprecated. Pleas
e use tf.compat.v1.get_default_session instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/t
ensorflow_backend.py:197: The name tf.ConfigProto is deprecated. Please use t
f.compat.v1.ConfigProto instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/t
ensorflow_backend.py:207: The name tf.global_variables is deprecated. Please u
se tf.compat.v1.global_variables instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/t
ensorflow_backend.py:216: The name tf.is_variable_initialized is deprecated. P
lease use tf.compat.v1.is_variable_initialized instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/t
ensorflow_backend.py:223: The name tf.variables_initializer is deprecated. Ple
ase use tf.compat.v1.variables_initializer instead.

10000/10000 [==============================] - 9s 937us/step - loss: 228.2652
- mean_squared_error: 144.7157
Epoch 2/20
10000/10000 [==============================] - 0s 5us/step - loss: 145.9072 -
mean_squared_error: 86.2181
Epoch 3/20
10000/10000 [==============================] - 0s 4us/step - loss: 133.0892 -
mean_squared_error: 68.1974
Epoch 4/20
10000/10000 [==============================] - 0s 5us/step - loss: 132.1757 -
mean_squared_error: 60.5807
Epoch 5/20
10000/10000 [==============================] - 0s 4us/step - loss: 130.2000 -
mean_squared_error: 65.4128
Epoch 6/20
10000/10000 [==============================] - 0s 4us/step - loss: 130.4460 -
mean_squared_error: 67.7295
Epoch 7/20
10000/10000 [==============================] - 0s 5us/step - loss: 130.0326 -
mean_squared_error: 65.5014
Epoch 8/20
10000/10000 [==============================] - 0s 5us/step - loss: 130.1651 -
mean_squared_error: 64.4970
Epoch 9/20
10000/10000 [==============================] - 0s 5us/step - loss: 130.0697 -
mean_squared_error: 65.3174
Epoch 10/20
10000/10000 [==============================] - 0s 5us/step - loss: 130.0714 -
mean_squared_error: 64.9093
Epoch 11/20
10000/10000 [==============================] - 0s 4us/step - loss: 130.0598 -
mean_squared_error: 64.9172
Epoch 12/20
10000/10000 [==============================] - 0s 5us/step - loss: 130.0663 -
mean_squared_error: 65.2315
Epoch 13/20
```

Out[8]: <keras.callbacks.History at 0x7f985cc692b0>

In [9]:
```
df_1 = df.copy()
df_1['X1'] = 1
df['Y_1'] = model.predict(df_1[['X1']])

df_0 = df.copy()
df_0['X1'] = 0
df['Y_0'] = model.predict(df_0[['X1']])
(df['Y_1']-df['Y_0']).mean()
```

Out[9]: 12.40583324432373

This approach achieves an estimate of 12.40 (3% bias). We can bootstrap the model estimates to obtain a confidence interval on the estimate. First, we must use repeat the process used to generate different estimates for the average treatment effects. We compute 200 models (Using Google Colab with GPU) and then start the bootstrap process.

In [14]:

```
ATEs = []
for i in range(200):
  model = Model(inputs=[x_in],outputs=[y])
  model.compile('adam',loss='mse',metrics=['mse'])
  model.fit(df[['X1']],df[['Y']], batch_size=1024,epochs=20,sample_weight=df['we
ight'])
  df['Y_1'] = model.predict(df_1[['X1']])
  df['Y_0'] = model.predict(df_0[['X1']])
  ATEs.append((df['Y_1']-df['Y_0']).mean())
  print(i)
```

Streaming output truncated to the last 5000 lines.

```
10000/10000 [==============================] - 0s 7us/step - loss: 130.1169 -
mean_squared_error: 64.7099
Epoch 3/20
10000/10000 [==============================] - 0s 7us/step - loss: 130.0505 -
mean_squared_error: 64.4149
Epoch 4/20
10000/10000 [==============================] - 0s 7us/step - loss: 130.0572 -
mean_squared_error: 65.5371
Epoch 5/20
10000/10000 [==============================] - 0s 7us/step - loss: 130.0959 -
mean_squared_error: 65.1198
Epoch 6/20
10000/10000 [==============================] - 0s 6us/step - loss: 130.0342 -
mean_squared_error: 65.0219
Epoch 7/20
10000/10000 [==============================] - 0s 7us/step - loss: 130.0754 -
mean_squared_error: 65.3923
Epoch 8/20
10000/10000 [==============================] - 0s 7us/step - loss: 130.1287 -
mean_squared_error: 64.7557
Epoch 9/20
10000/10000 [==============================] - 0s 7us/step - loss: 130.0622 -
mean_squared_error: 64.6625
Epoch 10/20
10000/10000 [==============================] - 0s 7us/step - loss: 130.0575 -
mean_squared_error: 64.8718
Epoch 11/20
10000/10000 [==============================] - 0s 7us/step - loss: 129.9978 -
mean_squared_error: 64.9602
Epoch 12/20
10000/10000 [==============================] - 0s 7us/step - loss: 130.0953 -
mean_squared_error: 64.9649
Epoch 13/20
10000/10000 [==============================] - 0s 7us/step - loss: 129.9928 -
mean_squared_error: 65.2802
Epoch 14/20
10000/10000 [==============================] - 0s 8us/step - loss: 130.0394 -
mean_squared_error: 65.5724
Epoch 15/20
10000/10000 [==============================] - 0s 7us/step - loss: 130.0109 -
mean_squared_error: 65.2700
Epoch 16/20
10000/10000 [==============================] - 0s 8us/step - loss: 130.0742 -
mean_squared_error: 64.6993
Epoch 17/20
10000/10000 [==============================] - 0s 7us/step - loss: 130.0081 -
mean_squared_error: 64.9193
Epoch 18/20
10000/10000 [==============================] - 0s 7us/step - loss: 130.0382 -
mean_squared_error: 65.5487
Epoch 19/20
10000/10000 [==============================] - 0s 7us/step - loss: 130.1117 -
mean_squared_error: 65.0628
Epoch 20/20
10000/10000 [==============================] - 0s 7us/step - loss: 130.0065 -
mean_squared_error: 64.8960
78
Epoch 1/20
10000/10000 [==============================] - 9s 863us/step - loss: 130.0599
- mean_squared_error: 65.3759
Epoch 2/20
10000/10000 [==============================] - 0s 7us/step - loss: 130.1503 -
mean_squared_error: 64.2156
Epoch 3/20
10000/10000 [==============================] - 0s 7us/step - loss: 130.0001 -
mean_squared_error: 65.0635
Epoch 4/20
10000/10000 [==============================] - 0s 7us/step - loss: 130.0547 -
```

We create a bootstrap with B = 1000 dataframes each containing 50 rows to construct a 0.95 confidence interval for the average ATE. Using the 0.025 and 0.975 bootstrap empirical quantiles we will derive the 0.95 confidence interval.

In [15]:
```
df_to_bootstrap = pd.DataFrame({'ATE':ATEs})

n_bootstrap = 1000 #Number of bootstrap samples to generate
bootstrap_sample_length = 50 #length of the bootstrap sample
bootstrap_averages = []

for b in range(n_bootstrap):
  boot_sample = df_to_bootstrap.sample(bootstrap_sample_length,replace=True)
  bootstrap_averages.append(boot_sample.mean())
bootstrap_frame = pd.DataFrame({'averages':bootstrap_averages})
print('bootstrap mean : ',bootstrap_frame['averages'].mean())
np.percentile(bootstrap_averages,[2.5,97.5])
```

```
bootstrap mean :  12.5043702949333
```

Out[15]:
```
array([12.4790669 , 12.52693337])
```

We obtain a 95% confidence interval of $[12.4790669, 12.52693337]$, meaning there is still some bias in our model.

(h) How might you make this last estimate doubly-robust?

To make the estimate doubly robust, we need to include the effects $Z \to X_1$ and $Z \to \ldots \to Y$ in our model (with $Z = \{U_1\}$). We incorporate the effect of Z on X_1 by considering the propensity score in the sample_weights of our feedforward network. We incorporate the effect of Z on Y by adding U_1 to the regressors as we fit the model.

In [17]:
```
x_in = Input(shape=(2,))
h1 = Dense(128,activation='tanh')(x_in)
h2 = Dense(128,activation='tanh')(h1)
h3 = Dense(128,activation='tanh')(h2)
h4 = Dense(128,activation='tanh')(h3)
y = Dense(1,activation='linear')(h4)


model = Model(inputs=[x_in],outputs=[y])
model.compile('adam',loss='mse',metrics=['mse'])
model.fit(df[['X1','U1']],df[['Y']], batch_size=1024,epochs=20,sample_weight=df
['weight'])
```

```
Epoch 1/20
10000/10000 [==============================] - 16s 2ms/step - loss: 214.9820 -
mean_squared_error: 133.0950
Epoch 2/20
10000/10000 [==============================] - 0s 10us/step - loss: 115.6110 -
mean_squared_error: 65.1606
Epoch 3/20
10000/10000 [==============================] - 0s 10us/step - loss: 110.9536 -
mean_squared_error: 55.2423
Epoch 4/20
10000/10000 [==============================] - 0s 9us/step - loss: 103.0778 -
mean_squared_error: 53.2681
Epoch 5/20
10000/10000 [==============================] - 0s 10us/step - loss: 101.1319 -
mean_squared_error: 54.2585
Epoch 6/20
10000/10000 [==============================] - 0s 10us/step - loss: 97.0194 -
mean_squared_error: 51.9368
Epoch 7/20
10000/10000 [==============================] - 0s 10us/step - loss: 94.6950 -
mean_squared_error: 49.6988
Epoch 8/20
10000/10000 [==============================] - 0s 10us/step - loss: 92.5094 -
mean_squared_error: 48.3083
Epoch 9/20
10000/10000 [==============================] - 0s 10us/step - loss: 91.6334 -
mean_squared_error: 47.4381
Epoch 10/20
10000/10000 [==============================] - 0s 11us/step - loss: 91.2421 -
mean_squared_error: 46.8858
Epoch 11/20
10000/10000 [==============================] - 0s 10us/step - loss: 89.6206 -
mean_squared_error: 45.7601
Epoch 12/20
10000/10000 [==============================] - 0s 9us/step - loss: 88.6412 - m
ean_squared_error: 45.0483
Epoch 13/20
10000/10000 [==============================] - 0s 9us/step - loss: 88.2344 - m
ean_squared_error: 44.6384
Epoch 14/20
10000/10000 [==============================] - 0s 9us/step - loss: 88.2450 - m
ean_squared_error: 44.4190
Epoch 15/20
10000/10000 [==============================] - 0s 8us/step - loss: 87.2834 - m
ean_squared_error: 43.9059
Epoch 16/20
10000/10000 [==============================] - 0s 8us/step - loss: 87.0212 - m
ean_squared_error: 43.6747
Epoch 17/20
10000/10000 [==============================] - 0s 9us/step - loss: 87.3583 - m
ean_squared_error: 43.6614
Epoch 18/20
10000/10000 [==============================] - 0s 9us/step - loss: 87.3181 - m
ean_squared_error: 43.6271
Epoch 19/20
10000/10000 [==============================] - 0s 9us/step - loss: 87.8175 - m
ean_squared_error: 43.5518
Epoch 20/20
10000/10000 [==============================] - 0s 9us/step - loss: 86.3453 - m
ean_squared_error: 43.3139
```

Out[17]: <keras.callbacks.History at 0x7f97f2dc2f60>

In [20]:
```
df['Y_1'] = model.predict(df_1[['X1','U1']])
df['Y_0'] = model.predict(df_0[['X1','U1']])
(df['Y_1']-df['Y_0']).mean()
```

Out[20]:
```
11.913871765136719
```

This approach yields an estimate of 11.91, less than 1% bias, our best model yet.

In [0]:
```

```