CSCI-S89, Introduction to Deep Learning
Homework 1 (problems 1 & 2)
Mark Carlebach
June 26, 2020

Problem 1 (20 points)
Please load 'Housing_Data.csv' - the housing data in Singapore. This data set is also available at https://www.kaggle.com/chenzhiliang/housing-data. Using Keras, build a Neural Network with one hidden layer which has two neurons, predict the price based on 'floorArea' and 'bedrooms'. Here, you can implement either batch or mini-batch gradient descent algorithm. Make sure to use appropriate loss and activation functions. Plot the train and test errors versus iteration step.

SOLUTION:

Attachments:

- carlebach_hw1_problem1.ipynb

Description of solution:

I read and clean the data, including dropping a few observations that have bad data or are outliers. I produce the following plot to show data prior to training.

I then scale the features, build the model and fit the model as shown here.

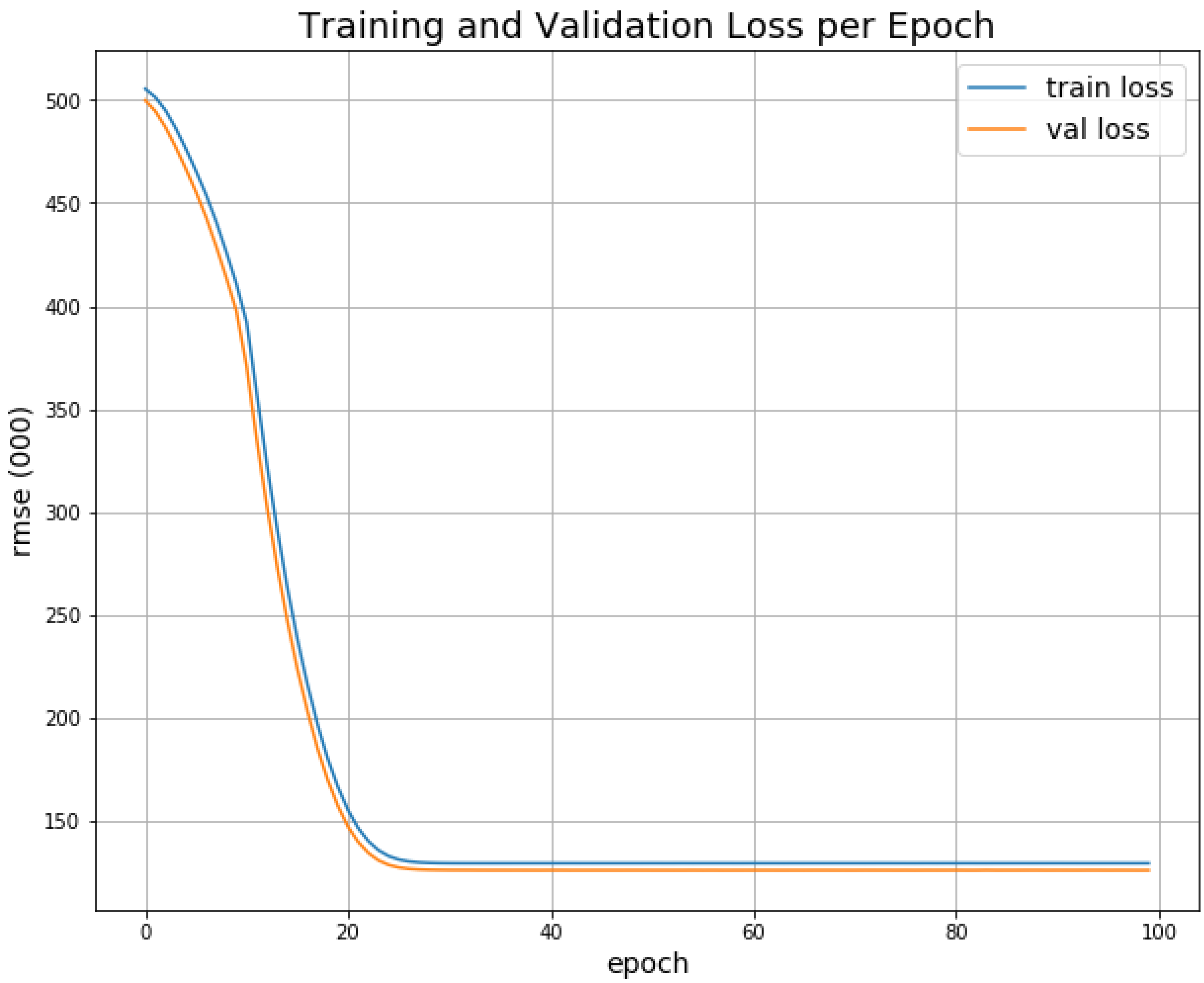
```
# Put data into X & y numpy arrays and scale
scaler = StandardScaler()
X = np.zeros([price.shape[0], 2])
X[:,0] = floor_area
X[:,1] = bedrooms
scaler.fit(X)
X = scaler.transform(X)
y = price / 1000

# Create train and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.2, random_state=89)
```

```
# Create NN model
model = models.Sequential()
model.add(layers.Dense(2, activation='relu', input_shape=(2,)))
model.add(layers.Dense(1, activation='linear'))
print(model.summary())

# Compile the model
opt = keras.optimizers.Adam(learning_rate=0.005)
model.compile(loss="mean_squared_error", optimizer=opt)

# Fit the model
epochs = 100
history = model.fit(X_train, y_train, epochs=epochs,
                    batch_size=128,
                    verbose=0,
                    validation_data=(X_test, y_test))
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 2)                 6
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 3
=================================================================
Total params: 9
Trainable params: 9
Non-trainable params: 0
_________________________________________________________________
None
```

I plot the training and validation loss here.

```
# Plot results
fig, ax = plt.subplots(1, 1, figsize=(10,8))
ax.set_title("Training and Validation Loss per Epoch", fontsize=18)
ax.set_xlabel("epoch", fontsize=14)
ax.set_ylabel("rmse (000)", fontsize=14)
ax.plot(np.array(history.history["loss"]) ** .5, label = "train loss")
ax.plot(np.array(history.history["val_loss"]) ** .5, label = "val loss")
ax.legend(fontsize=14)
ax.grid(True)
plt.show()
```



The model does not seem to do a great job based on the size of the RMSE compared to the mean of the price data.

```
price_mean = price.mean()
print(f"Mean price = ${price_mean :6.0f}")

rmse = min(history.history["val_loss"]) ** 0.5
print(f"RMSE = ${rmse :3.0f}000")
```

```
Mean price = $481442
RMSE = $126000
```

Commnet:

- The RMSE on validation data is pretty high relative to the mean price of a home.
- This does not seem like a very good model.

Problem 2 (45 points)
Construct the Neural Network you used in Problem 1 without the use of packages. Implement the Backpropagation "manually" and plot the test and train errors.

SOLUTION:

Attachments:

- carlebach_hw2_problem1.ipynb
- myTorch.py (should be in same directory as notebook so it is importable)

Description of solution:

NOTE: Almost identical solution as to Problem 1, except I import SequentialModel from myTorch.py rather than from Keras. I explain classes in myTorch.py below.

I read and clean the data, including dropping a few observations that have bad data or are outliers. I produce the following plot to show data prior to training.

I then scale the features, build the model and fit the model as shown here.

```python
# Put data into X & y numpy arrays and scale
scaler = StandardScaler()
X = np.zeros([price.shape[0], 2])
X[:,0] = floor_area
X[:,1] = bedrooms
scaler.fit(X)
X = scaler.transform(X)
y = price / 1000

# Create train and test data
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=.2, random_state=89)
```

```python
# Create NN model using myTorch implementation
model = SequentialModel(n_inputs=2, hidden_dim=2)
print(model.summary())

# Fit the model
history = model.fit(X_train=X_train, y_train=y_train,
                    X_val=X_val, y_val=y_val,
                    n_epochs=100, lr=1e-6,
                    batch_size=128, verbose=False)
```

```
myTorch Model Summary:
>>fully connected
  input_dim=2 :: hidden_layers=1 :: hidden_dim=2 :: output_dim=1
```

The main difference between my solution to Problem 2 and Problem 1 is this line:

model = SequentialModel(n_inputs-2, hidden_dim=2)

SequentialModel is imported from myTorch.py and performs forward and back propagation calculations. myTorch.py is a module with 3 classes that are meant to mimic the API of Keras for a very limited neural network. Here is a description of those classes with their attributes and methods:

class Node:

"""
Basic computational building block of neural network

Each node contains the following:
- node_id: unique id
- inputs: list of values passed in the forward direction
- weights_in: list of weights for each incoming connection (including bias)
- z: sum of (inputs * weights_in)
- u: act(z)
- weights_out: list of weights for each outgoing connection
- fprime: for relu, 1 if z > 0 else 0
- error: partial L wrt u
- partials: partials L wrt each weight_in

Methods:
  - forward(): calculates z & u based on inputs and weights
  - backward(): calculates fprime for node and error for node with fprime and error from node to right
"""

class FC_Layer:

"""
Layer is list of nodes. This is a fully connected layer so each node has weight from each node in preceding layer.

Each layer contains the following:
  - layer_id: unique id
  - nodes: list of nnodes nodes (initialized with random weights) where each node as weight from bias and each node of prior input layer (e.g., fully connected)

Methods:
  - forward(): calculates z & u for all nodes in layer, based on inputs and weights
  - backward(): calculates fprime and error for all nodes in layer
"""

class SequentialModel:

"""
A list of fully connected layers
  - The first layer has n_inputs nodes
  - There is only 1 single hidden layer (limitation of this implementation) with hidden_dim nodes
  - The last layer has 1 output node (limitation of this implementation)
"""

If you want to see the code that implements these classes and their methods, you can look in myTorch.py.   I am not sure it helps to cut and paste more from that file into this document.

After fitting the model (as with Problem 1), I plot the training and validation loss. The results are very similar to when I did the fitting in Keras.

As in Problem 1, the model does not seem to do a great job based on the size of the RMSE compared to the mean of the price data.

```
price_mean = price.mean()
print(f"Mean price = ${price_mean :6.0f}")

rmse = min(history["val_loss"]) ** 0.5
print(f"RMSE = ${rmse :3.0f}000")
```

```
Mean price = $481442
RMSE = $125000
```

Commnet:

- The RMSE on validation data is pretty high relative to the mean price of a home.
- This does not seem like a very good model.