IE 8990

Spring 2022

Homework #3

Due Date: 04/07/2022 5PM CST

Submission: Please put your answer and code in a PDF file and upload on Canvas

Q1. In LSTM model, why sigmoid or tanh are used? Can we use ReLU to replace tanh?

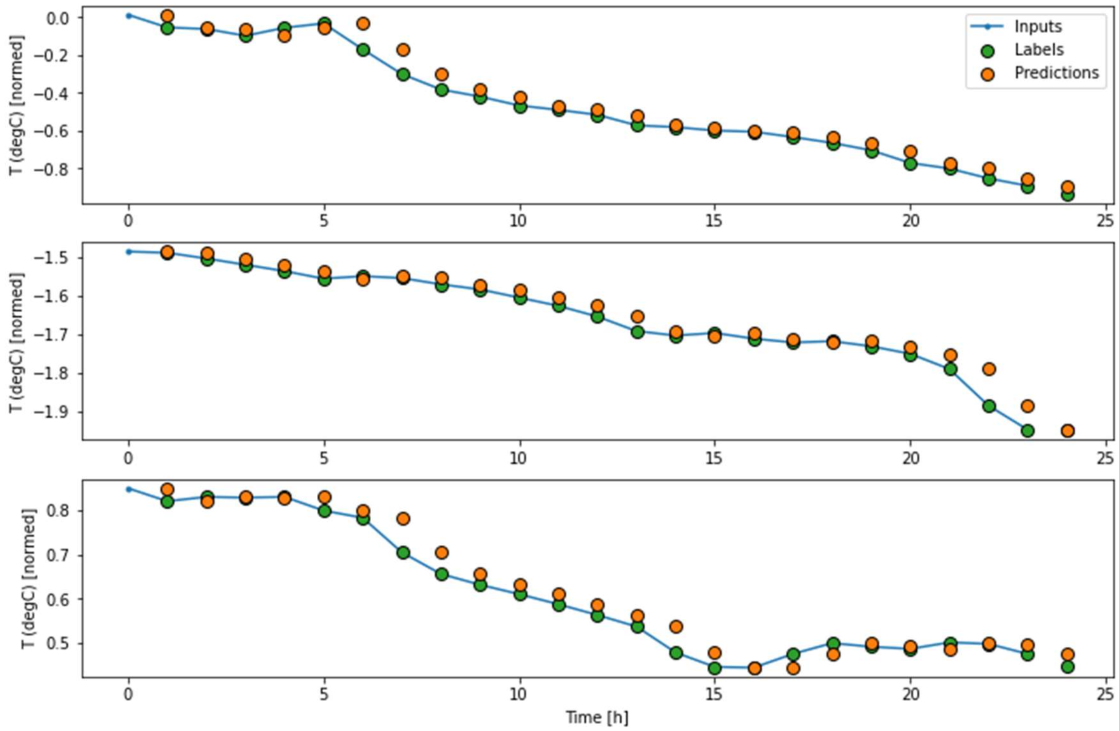
- The sigmoid and hyperbolic tangent functions are used to have binary values for decision making. The sigmoid sections are used to retain somewhere between 0 and 1 of the previous state. The tanh function is used to allow for negative values between -1 and 1.
- The ReLU function would not work since the activation function needs to be bounded. If the forget gate can be weighted past 1 then it is technically remembering more than 100% of the information. The negative values of the hyperbolic tangent are also important when highlighting the main issues of the functions.

Q2. For the HW2 CIFAR-10 Base Model (provided in attached Jupyter Notebook file), let's fix the epochs = 10. Please modify the model structure to improve the model performance based on the tips we discussed in DL 10 (target: test error lower than 0.25 in 10 epochs training). Discuss your approach.

- Through some experimentation it seemed that the best results came from simply adding more convolutional layers and more dense layers at the end was the biggest factor. Early stopping is unnecessary since we are only training for 10 epochs, and it would need to stop around 50. Regularization has a positive effect on the testing data; however, batch normalization was already in the dataset. Drop-out showed to have a significant negative effect, this is probably due to how small our neural network is. ReLu seemed to be the best activation function for this model in the hidden layers. Weight initialization, I assumed would have a huge impact on the accuracy since we are only running 10 epochs, but in my experiments, it showed almost zero change in the output. Finally, the adaptive learning rate, loss function, and optimizer only showed worse results when I attempted to change them (or add them in the case of the adaptive learning rate.)

Q3. Based on the jena climate 2009 01.csv data: Please develop a LSTM model to predict the next 24 hours' temperature (Celsius) based on the previous 24 hours' information. (Note: you can use all 14 climate features or part of them). Hint: here is a link might be useful: link). Please plot your predicted value and the true value in one plot.

- I assumed this is not a multi-window problem since the model needs to be ran with the previous true label and not the previous predicted label. I also assumed the dataset should not be touched for this experiment, so I did not arrange the data other than putting into a TensorFlow dataset. The code is at the bottom of this document, and several functions used to create the model are found on the TensorFlow website since they are currently not functions in TensorFlow.


Inputs
Labels
Predictions
T (degC) [normed]
Time [h]

Q4. For the following models AlexNet, ZFNet, VGG, GoogLeNet, ResNet, MobileNet, DenseNet, EfficientNet. Discuss the advantages and disadvantages of each model. Discuss the motivations of how each model was developed

- AlexNet is a convolutional neural network was created to compete in a competition for the ImageNet dataset. Its main advantage was pushing new ideas at the time such as being one of the first GPU assisted neural networks. This allowed the model to be very complicated at little to no risk for how computationally expensive the model was. The disadvantage of this model is that it is specially designed to handle the ImageNet dataset and needs several adjustments to be used on other datasets. The fully interconnected pooling layers means this model is very computationally expensive despite the use of GPU's.
- ZFNet was created to open the world to inner architecture of neural networks, specifically feature extraction in CNN's. The model attempts to deconvolute the image after convolutions. This allows for feature extraction to keep the original size and shape of the feature. This has several positives if performed correctly especially when working with human recognition data. This is due to the nature of feature differences being hard to differentiate when convoluted. The main disadvantage is that some datasets do not need to be deconvoluted so this very computationally expensive neural network would be running unnecessarily.
- VGG is a convolutional neural network designed to run the ImageNet dataset. It is very similar to the AlexNet; however, has more convolutions. This model is strictly better than the AlexNet model for this dataset. That is its major disadvantage since this model was designed for a very specific dataset.
- GoogLeNet is a model designed to be an improvement for the inception model and was made for the ILSVRC 2014 competition. This is a very complicated model that adds several layers of convolutions. This model shows great results but is one of the most computationally expensive models to train on this list.
- ResNet is an improvement of the AlexNet model made in 2015. This model was made to fix the vanishing gradient problem. It achieves this through adding the "identity shortcut connection". The major advantage to adding this skip connection in the neural network is that it significantly reduces the vanishing gradient problem; however, it is very difficult to implement properly. The skip connection can allow for some neurons to not be trained properly and overall making the network mostly incapable of learning. This is not too big of an issue if implemented in very large datasets.
- MobileNet is the first TensorFlow model capable of running on mobile devices. It achieves this by using depth-wise separable convolutions which significantly reduces the trainable parameters of the neural network. The main idea is that you get more accuracy than you normally would for a less taxing model, which is great for running neural networks on devices such as an iPhone. The main disadvantage is that its accuracy is not to par with modern complicated neural networks such as GoogLeNet.
- DenseNet is less of a specific model and more of an overall architecture design. The objective of the DenseNet is to combine every layer by having all layers connected at each node. Unlike the ResNet which gave the neural network to skip the node. This has shown to have major accuracy increases across several different datasets with Convolutional layers. Though these models become exponentially more computationally expensive as the number of layers are increased.

- EfficientNet is similar to the DenseNet in the aspect of it's not a specific neural network but more of a design philosophy. The main idea behind the EfficientNet is that the model is sized down using matrix operations. This could be used to scale down models, but most papers focus on its ability to scale up previously made models. The MobileNet and ResNet models have shown great improvement after being scaled through this method. The only negative that I can really see is that finding the perfect model size through this scaling will be very computationally expensive.

Question 2 Hw3

April 18, 2022

[1]:
```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
```

[2]:
```
# Change to Markdown if GPU is not supported.



import os

os.environ["TF_CPP_MIN_LOG_LEVEL"] = "2"

physical_devices = tf.config.list_physical_devices("GPU")
tf.config.experimental.set_memory_growth(physical_devices[0], True)
```

[3]:
```
# You don't need to change this session
um_classes = 10
input_shape = (32, 32, 3)

(X_train, y_train), (X_test, y_test) = keras.datasets.cifar10.load_data()

print("x_train shape: {} - y_train shape: {}".format(X_train.shape,y_train.
 ↪shape))
print("x_test shape: {} - y_test shape: {}".format(X_test.shape,y_test.shape))

# Scale images to the [0, 1] range
X_train = X_train.astype("float32") / 255
X_test = X_test.astype("float32") / 255
# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, um_classes)
y_test = keras.utils.to_categorical(y_test, um_classes)
```

```
x_train shape: (50000, 32, 32, 3) - y_train shape: (50000, 1)
x_test shape: (10000, 32, 32, 3) - y_test shape: (10000, 1)
```

[46]:
```
# Designing the Custom Model


inputs = keras.Input(shape=(32, 32, 3))

x=layers.Conv2D(64, kernel_size=(3, 3))(inputs)
x=layers.Activation("relu")(x)
x=layers.BatchNormalization()(x)

x=layers.MaxPooling2D(pool_size=(2, 2))(x)
x=layers.Conv2D(128, kernel_size=(3, 3))(x)
x=layers.Activation("relu")(x)
x=layers.BatchNormalization()(x)

x=layers.Conv2D(256, kernel_size=(3, 3))(x)
x=layers.Activation("relu")(x)
x=layers.BatchNormalization()(x)

x=layers.Conv2D(512, kernel_size=(3, 3)(x)
x=layers.Activation("relu")(x)
x=layers.BatchNormalization()(x)

x=layers.MaxPooling2D(pool_size=(2, 2))(x)
x=layers.Flatten()(x)

x=layers.Dense(512, activation='relu')(x)
x=layers.Dense(256, activation='relu')(x)
x=layers.Dense(128, activation='relu')(x)
x=layers.Dense(64, activation='relu')(x)
x=layers.Dense(32, activation='relu')(x)

outputs=layers.Dense(um_classes, activation="softmax")(x)
```

[50]:
```
# Compiling the Model


model=keras.Model(inputs,outputs)
model.compile(loss="categorical_crossentropy", optimizer="adam",␣
 ↪metrics=["accuracy"])
model.summary()
```

```
Model: "model_11"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_12 (InputLayer)       [(None, 32, 32, 3)]       0
```

```
conv2d_43 (Conv2D)          (None, 30, 30, 64)        1792

activation_46 (Activation)  (None, 30, 30, 64)        0

batch_normalization_40 (Bat  (None, 30, 30, 64)       256
chNormalization)

max_pooling2d_25 (MaxPoolin  (None, 15, 15, 64)       0
g2D)

conv2d_44 (Conv2D)          (None, 13, 13, 128)       73856

activation_47 (Activation)  (None, 13, 13, 128)       0

batch_normalization_41 (Bat  (None, 13, 13, 128)      512
chNormalization)

conv2d_45 (Conv2D)          (None, 11, 11, 256)       295168

activation_48 (Activation)  (None, 11, 11, 256)       0

batch_normalization_42 (Bat  (None, 11, 11, 256)      1024
chNormalization)

conv2d_46 (Conv2D)          (None, 9, 9, 512)         1180160

activation_49 (Activation)  (None, 9, 9, 512)         0

batch_normalization_43 (Bat  (None, 9, 9, 512)        2048
chNormalization)

max_pooling2d_26 (MaxPoolin  (None, 4, 4, 512)        0
g2D)

flatten_11 (Flatten)        (None, 8192)              0

dense_62 (Dense)            (None, 512)               4194816

dense_63 (Dense)            (None, 256)               131328

dense_64 (Dense)            (None, 128)               32896

dense_65 (Dense)            (None, 64)                8256

dense_66 (Dense)            (None, 32)                2080

dense_67 (Dense)            (None, 10)                330
```

```
=================================================================
Total params: 5,924,522
Trainable params: 5,922,602
Non-trainable params: 1,920
_________________________________________________________________
```

[51]:
```
# Fitting the Model


history = model.fit(X_train, y_train, epochs=10, batch_size=32,␣
 ↪validation_split=0.2, verbose=1)
```

```
Epoch 1/10
1250/1250 [==============================] - 41s 32ms/step - loss: 0.1290 -
accuracy: 0.9605 - val_loss: 1.0593 - val_accuracy: 0.7571
Epoch 2/10
1250/1250 [==============================] - 30s 24ms/step - loss: 0.1041 -
accuracy: 0.9686 - val_loss: 1.2016 - val_accuracy: 0.7411
Epoch 3/10
1250/1250 [==============================] - 33s 26ms/step - loss: 0.0948 -
accuracy: 0.9719 - val_loss: 1.4311 - val_accuracy: 0.7143
Epoch 4/10
1250/1250 [==============================] - 30s 24ms/step - loss: 0.0928 -
accuracy: 0.9721 - val_loss: 1.1589 - val_accuracy: 0.7588
Epoch 5/10
1250/1250 [==============================] - 30s 24ms/step - loss: 0.0858 -
accuracy: 0.9746 - val_loss: 1.1024 - val_accuracy: 0.7541
Epoch 6/10
1250/1250 [==============================] - 29s 23ms/step - loss: 0.0832 -
accuracy: 0.9756 - val_loss: 1.1593 - val_accuracy: 0.7472
Epoch 7/10
1250/1250 [==============================] - 33s 26ms/step - loss: 0.0674 -
accuracy: 0.9810 - val_loss: 1.0693 - val_accuracy: 0.7667
Epoch 8/10
1250/1250 [==============================] - 23s 18ms/step - loss: 0.0775 -
accuracy: 0.9781 - val_loss: 1.2517 - val_accuracy: 0.7567
Epoch 9/10
1250/1250 [==============================] - 23s 18ms/step - loss: 0.0720 -
accuracy: 0.9793 - val_loss: 1.1480 - val_accuracy: 0.7464
Epoch 10/10
1250/1250 [==============================] - 23s 18ms/step - loss: 0.0667 -
accuracy: 0.9808 - val_loss: 1.2986 - val_accuracy: 0.7397
```

[52]:
```
# Final Scores


score = model.evaluate(X_test, y_test, verbose=0)
```

```python
print("Test loss:", score[0])
print("Test error:", 1-score[1])
```

```
Test loss: 1.3165239095687866
Test error: 0.25950002670288086
```

Question 3 Hw3

April 18, 2022

1 Question 3

1.1 Importing Libraries

[1]:
```
import os
import datetime

import IPython
import IPython.display
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import tensorflow as tf

mpl.rcParams['figure.figsize'] = (8, 6)
mpl.rcParams['axes.grid'] = False
```

1.2 Preparing Data

[2]:
```
# Loading Data


csv_path = "jena_climate_2009_01.csv"
```

[3]:
```
# Creating Dataframe of Loaded Data

df = pd.read_csv(csv_path)

df.head()
```

[3]:
```
             Date Time  p (mbar)  T (degC)  Tpot (K)  Tdew (degC)  rh (%)  \
0  01.01.2009 00:00:00    996.52     -8.02    265.40        -8.90    93.3
1  01.01.2009 00:10:00    996.52     -8.02    265.40        -8.90    93.3
2  01.01.2009 00:20:00    996.57     -8.41    265.01        -9.28    93.4
3  01.01.2009 00:30:00    996.53     -8.51    264.91        -9.31    93.9
```

```
4  01.01.2009 00:40:00    996.51     -8.31    265.12        -9.07    94.2

   VPmax (mbar)  VPact (mbar)  VPdef (mbar)  sh (g/kg)  H2OC (mmol/mol)  \
0          3.33          3.11          0.22       1.94             3.12
1          3.33          3.11          0.22       1.94             3.12
2          3.23          3.02          0.21       1.89             3.03
3          3.21          3.01          0.20       1.88             3.02
4          3.26          3.07          0.19       1.92             3.08

   rho (g/m**3)  wv (m/s)  max. wv (m/s)  wd (deg)
0       1307.75      1.03           1.75     152.3
1       1307.75      1.03           1.75     152.3
2       1309.80      0.72           1.50     136.1
3       1310.24      0.19           0.63     171.6
4       1309.19      0.34           0.50     198.0
```

[4]:

```
# Splitting the Data

column_indices = {name: i for i, name in enumerate(df.columns)}

n = len(df)
train_df = df[0:int(n*0.7)]
val_df = df[int(n*0.7):int(n*0.9)]
test_df = df[int(n*0.9):]

num_features = df.shape[1]
```

[5]:

```
# Normalizing the Data

train_mean = train_df.mean()
train_std = train_df.std()

train_df = (train_df - train_mean) / train_std
val_df = (val_df - train_mean) / train_std
test_df = (test_df - train_mean) / train_std

# I recieve errors but it does normalize data
```

```
C:\Users\PRESTO~1\AppData\Local\Temp/ipykernel_53172/3366875348.py:3:
FutureWarning: Dropping of nuisance columns in DataFrame reductions (with
'numeric_only=None') is deprecated; in a future version this will raise
TypeError.  Select only valid columns before calling the reduction.
  train_mean = train_df.mean()
C:\Users\PRESTO~1\AppData\Local\Temp/ipykernel_53172/3366875348.py:4:
FutureWarning: Dropping of nuisance columns in DataFrame reductions (with
'numeric_only=None') is deprecated; in a future version this will raise
TypeError.  Select only valid columns before calling the reduction.
```

```
train_std = train_df.std()
```

1.3 Defining Functions

Most of the functions below are taken from different websites in order to perform this task. These are the same functions that I am using in my experimental testing for the LSTM update.

[6]:
```
# Defining the WindowGenerator, this function will be called later to define␣
↪how much
# information we want to make the LSTM desicion based off of.


class WindowGenerator():
  def __init__(self, input_width, label_width, shift,
               train_df=train_df, val_df=val_df, test_df=test_df,
               label_columns=None):
    # Store the raw data.
    self.train_df = train_df
    self.val_df = val_df
    self.test_df = test_df

    # Work out the label column indices.
    self.label_columns = label_columns
    if label_columns is not None:
      self.label_columns_indices = {name: i for i, name in
                                    enumerate(label_columns)}
    self.column_indices = {name: i for i, name in
                           enumerate(train_df.columns)}

    # Work out the window parameters.
    self.input_width = input_width
    self.label_width = label_width
    self.shift = shift

    self.total_window_size = input_width + shift

    self.input_slice = slice(0, input_width)
    self.input_indices = np.arange(self.total_window_size)[self.input_slice]

    self.label_start = self.total_window_size - self.label_width
    self.labels_slice = slice(self.label_start, None)
    self.label_indices = np.arange(self.total_window_size)[self.labels_slice]

  def __repr__(self):
    return '\n'.join([
        f'Total window size: {self.total_window_size}',
        f'Input indices: {self.input_indices}',
        f'Label indices: {self.label_indices}',
```

```
        f'Label column name(s): {self.label_columns}'])
```

[7]:
```
# This is called to split the individual windows into what is defined.



def split_window(self, features):
  inputs = features[:, self.input_slice, :]
  labels = features[:, self.labels_slice, :]
  if self.label_columns is not None:
    labels = tf.stack(
        [labels[:, :, self.column_indices[name]] for name in self.
 ↪label_columns],
        axis=-1)

  # Slicing doesn't preserve static shape information, so set the shapes
  # manually. This way the `tf.data.Datasets` are easier to inspect.
  inputs.set_shape([None, self.input_width, None])
  labels.set_shape([None, self.label_width, None])

  return inputs, labels

WindowGenerator.split_window = split_window
```

[8]:
```
# This is how we will plot the results



def plot(self, model=None, plot_col='T (degC)', max_subplots=3):
  inputs, labels = self.example
  plt.figure(figsize=(12, 8))
  plot_col_index = self.column_indices[plot_col]
  max_n = min(max_subplots, len(inputs))
  for n in range(max_n):
    plt.subplot(max_n, 1, n+1)
    plt.ylabel(f'{plot_col} [normed]')
    plt.plot(self.input_indices, inputs[n, :, plot_col_index],
             label='Inputs', marker='.', zorder=-10)

    if self.label_columns:
      label_col_index = self.label_columns_indices.get(plot_col, None)
    else:
      label_col_index = plot_col_index

    if label_col_index is None:
      continue

    plt.scatter(self.label_indices, labels[n, :, label_col_index],
                edgecolors='k', label='Labels', c='#2ca02c', s=64)
```

```
    if model is not None:
      predictions = model(inputs)
      plt.scatter(self.label_indices, predictions[n, :, label_col_index],
                  edgecolors='k', label='Predictions',
                  c='#ff7f0e', s=64)

    if n == 0:
      plt.legend()

  plt.xlabel('Time [h]')

WindowGenerator.plot = plot
```

[9]:
```
# This function is used to make the data into a Tensorflow time-series based␣
↪dataset. This
# needs to be changed for every different dataset.


def make_dataset(self, data):
  data = np.array(data, dtype=np.float32)
  ds = tf.keras.utils.timeseries_dataset_from_array(
      data=data,
      targets=None,
      sequence_length=self.total_window_size,
      sequence_stride=1,
      shuffle=True,
      batch_size=32,)

  ds = ds.map(self.split_window)

  return ds

WindowGenerator.make_dataset = make_dataset
```

1.4 Preparing the Data

[10]:
```
# Creating Tensorflow Datasets

@property
def train(self):
  return self.make_dataset(self.train_df)

@property
def val(self):
  return self.make_dataset(self.val_df)

@property
```

```
def test(self):
  return self.make_dataset(self.test_df)

@property
def example(self):
  """Get and cache an example batch of `inputs, labels` for plotting."""
  result = getattr(self, '_example', None)
  if result is None:
    # No example batch was found, so get one from the `.train` dataset
    result = next(iter(self.train))
    # And cache it for next time
    self._example = result
  return result

WindowGenerator.train = train
WindowGenerator.val = val
WindowGenerator.test = test
WindowGenerator.example = example
```

1.5 Preparing the Model

[11]:
```
# Running the Model with Specifications

wide_window = WindowGenerator(
    input_width=24, label_width=24, shift=1,
    label_columns=['T (degC)'])

wide_window
```

[11]:
```
Total window size: 25
Input indices: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
21 22 23]
Label indices: [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21
22 23 24]
Label column name(s): ['T (degC)']
```

[12]:
```
# Creating the Model

class Baseline(tf.keras.Model):
  def __init__(self, label_index=None):
    super().__init__()
    self.label_index = label_index

  def call(self, inputs):
    if self.label_index is None:
      return inputs
    result = inputs[:, :, self.label_index]
```

```
    return result[:, :, tf.newaxis]
```

[13]:
```
# Running the Model

baseline = Baseline(label_index=column_indices['T (degC)'])

baseline.compile(loss=tf.losses.MeanSquaredError(),
                 metrics=[tf.metrics.MeanAbsoluteError()])

val_performance = {}
performance = {}
val_performance['Baseline'] = baseline.evaluate(wide_window.val)
performance['Baseline'] = baseline.evaluate(wide_window.test, verbose=0)
```

```
28/28 [==============================] - 1s 8ms/step - loss: 6.5680e-04 -
mean_absolute_error: 0.0157
```

1.6 Plot

[14]:
```
# Plotting the Data

wide_window.plot(baseline)
```