McMaster University, Physics 3G03 March 18 - April 8 2021
http://avenue.mcmaster.ca/

Homework 5 : Convolution Networks, Non-Linear Regression, Model Tuning and Tree Regression

You should submit source a python notebook along with a pdf of the notebook for all exercises.

Due: Thursday April 8 2021, 11:30pm on avenue, with a grace period till the following Monday at 11:30pm

☛**Reading:** Parts of the first lectures are based on *The Hundred Page Machine Learning Book* by A. Burkov. Found at `http://themlbook.com`. Since then we have gone through Chapters 2,3,4,5,6,7,8,10,11,13 and parts of App A,B in *Machine Learning Refined*. An early version of *Machine Learning Refined* is available at `https://github.com/jermwatt/machine_learning_refined` We have also covered parts of *Hands-On ML* by A. Géron. Currently we're finishing Chapter 11 in *Machine Learning Refined* and we will continue with Chapter 14.

Exercise 1 *Convolution Networks - Vertical and Horizontal Art*

In order to better understand convolution networks we will in this exercise generate some toy images and then analyze them using a convolution network.

Let us start by generating some imgages that contain a number `stripe_nr` of either horizontal or vertical stripes of random length at random positions. We can do that using the following function:

```
def generate_stripe_image(size, stripe_nr, vertical = True):
  img=np.zeros((size,size,1),dtype="uint8")
  for i in range(0,stripe_nr):
    x,y = np.random.randint(0,size,2)
    l  = np.int(np.random.randint(y,size,1))
    if (vertical):
      img[y:l,x,0]=255
    else:
      img[x,y:l,0]=255
  return img
```

You can have a look at the images generated by using:

```
img=generate_stripe_image(50,10, vertical=True)
plt.imshow(img[:,:,0],cmap='gray')
```

Note that we're only using the first channel in this image.

1.1 As in the example above, let us decide to work with 50×50 images each with 10 random stripes. Use the above function to generate 1,000 training images and 1,000 validation images `X_train, X_val`, each with exactly 500 images with horizonal stripes and 500 images with vertical stripes. Normalize the two sets.

1.2 Create labels `Y_train, Y_val` with the correct labels for the training and validation images. For instance, use 0 for images with vertical stripes and 1 for images with horizontal stripes.

In the following we shall use one-hot encoding, so convert `Y_train, Y_val` to one-hot format using the Tensorflow/Keras function `to_categorical`.

1.3 Use Tensorflow/Keras to build a convolution network with a few layers: a 5x5 kernel convolution layer with linear activation and a single filter, a max pooling 2D layer with `pool_size=50`, and a dense layer with 2 neurons activated with softmax. Compile it using:

```
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

How many parameters do you have ?

1.4 We can now use the one-hot encoded data to train the model. Let's do that using:

```
history=model.fit(X_train, Y_train,
                  validation_data=(X_val,Y_val),
                  batch_size=64,
                  epochs=50,
                  verbose=1,
                  shuffle=True)
```

Plot the model and validation accuracy as a function of the epoch. Make another plot of the model and validation loss versus the epoch. These are accessible as for instance `history.history['val_loss']` etc.

1.5 If things are working correctly the network should have found a kernel that correctly distinguishes between horizontal and vertical stripes. You can access the trained kernel using `my_kernel=model.get_weights()[0]`. Make a grey scale plot of the kernel. Comment on the visual appearance of the kernel. Does it seem reasonable that it is capable of distinguishing between horizonal and vertical stripes. Explain.

1.6 Experiment with your network. What happens if you use a relu activation instead of a linear activation in the convolution layer ? What happens if you use average pooling instead of max pooling ? What happens if you use a different kernel size.

Exercise 2 *Non-linear Regression*

In this exercise we work through Example 10.3 in the book based on the dataset `'multiple_sine_waves.csv'`. We want to fit the data to a set of $B = 2$ parameterized sinusoidal feature transformations:

$$\begin{aligned} f_1(\vec{x}) &= \sin(w_{1,0} + w_{1,1}x_1 + w_{1,2}x_2) \\ f_2(\vec{x}) &= \sin(w_{2,0} + w_{2,1}x_1 + w_{2,2}x_2) \end{aligned} \tag{1}$$

We will implement this exercise fully in python. As explained in section 10.2.3 it is useful to define the following functions:

```
# feature transformation
def feature_transforms(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return np.sin(a).T

def model(x,w):
    # feature transformation
    f = feature_transforms(x,w[0])

    # compute linear combination and return
    a = w[1][0] + np.dot(f.T,w[1][1:])
    return a.T
```

Use gradient descent with `max_its = 2000; alpha_choice = 10**(0);` and a least squares cost to determine the most optimal parameters with minimum cost. List the optimal parameters. Make a plot of the cost function versus iteration. (It's not required, but if you like you can then try to reproduce the plots in Figure 10.7, which are plots of what exactly ?)

Exercise 3 *Model Tuning*

Let's return to the MNIST data set of hand-written data that we have worked with previously. We want to experiment with how we best tune a model to get the most optimal version. To that end let us consider the following classifier built in Tensorflow/Keras

```
from tensorflow.keras.datasets import mnist
import numpy as np
from tensorflow import keras
from tensorflow.keras import layers

(train_images, train_labels), _ = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

alpha = 1.
model = keras.Sequential([
  layers.Dense(512, activation='relu'),
  layers.Dense(10, activation='softmax')
])
model.compile(optimizer=keras.optimizers.RMSprop(alpha),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
history=model.fit(train_images, train_labels,
          epochs=10,
          batch_size=128,
          validation_split=0.2)
```

3.1 From the history plot `'val_loss'` versus the epochs for different training rates, $\alpha = 1., 0.5, 0.1, 0.01$ and 0.001. The loss is in this case defined from the categorical crossentropy while the accuracy is the percentage of correct predictions. If you're running this in a jupyter notebooks you may need to clear the output or run the different values of α in separate notebooks. What happens as you decrease the value of α ? What can you say about the first value of $\alpha = 1.0$? Explain your results.

3.2 Now let us look at a very small model just doing logistic regression on the MNIST data:

```
model = keras.Sequential([layers.Dense(10, activation='softmax')])
model.compile(optimizer='rmsprop',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
history_small_model = model.fit(
    train_images, train_labels,
    epochs=50,
    batch_size=128,
    validation_split=0.2)
```

Note that, if we don't specify any learning rate then the `'rmsprop'` optimizer defaults to a learning rate of 0.001. You should again plot the `'val_loss'` versus the epochs. Is there a clear minimum in the validation loss indicating where we start to overfit ?

3.3 Now let us increase the complexity of the model by putting in hidden layers:

```
model = keras.Sequential([
  layers.Dense(96, activation='relu'),
  layers.Dense(96, activation='relu'),
  layers.Dense(10, activation='softmax'),
])
model.compile(optimizer='rmsprop',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
history_large_model = model.fit(
    train_images, train_labels,
    epochs=50,
    batch_size=128,
    validation_split=0.2)
```

Repeat the plot of the `'val_loss'` versus the epochs. Is there in this case a clear minimum with a well defined region of overfitting ? Comment on your observations.

☛**Comment:** Keras has a built in tuner which you can read about here:
`https://www.tensorflow.org/tutorials/keras/keras_tuner`
No need to use it here.

Exercise 4 *Boosting Tree Regression*

Scikit-learn has a powerful tree regressor built in. In this exercise we will verify that it is actually working correctly by explicitly implementing it in python and plotting the results together with the output from Scikit-learn. That sounds like a lot of work but we will simplify things by first of all focusing on a one-dimensional model and secondly *assuming that we only need to know the result of the regression in the points x_p.* The model predicted by Scikit-learn can be evaluated at any x but that becomes a little to much book keeping for our purposes. Following the Scikit-learn manual we generate noisy data for $\sin(x)$ and perform a tree regression to a `max_depth=2` and `max_depth=5` as follows:

```
# Create a random dataset
rng = np.random.RandomState(1)
X = np.sort(6.3 * rng.rand(100, 1), axis=0)
y = np.sin(X).ravel()
y += 0.3 * (0.5 - rng.rand(len(X)))
# Fit regression model
regr_1 = DecisionTreeRegressor(max_depth=2)
regr_2 = DecisionTreeRegressor(max_depth=5)
regr_1.fit(X, y)
regr_2.fit(X, y)

# Predict
y_1 = regr_1.predict(X)
y_2 = regr_2.predict(X)
```

In this example we are asking Scikit-learn for 2 tree regressions to a maximum depth of 2 and 5, respectively.

Suppose, we have noisy data $\{x_p, y_p\}$ with x_p sorted along the x-axis. Then, as discussed in class and in chapter 14, the splitting points for the tree stumps will be *between* two values of x_s and x_{s+1}. The value for the leaves on either side is simply the average of the y_p's on either side. In order to determine the optimal split we can use a least squared cost. We can then write a function that recursively calls it self on smaller and smaller intervals. To simply things we can assume all the x_p and y_p are distinct. Implement a simple version of a tree regressor that can perform a tree regression to a specified maximum depth. Make sure that you correctly handle the case where an interval contains just a single point. Plot your results along with the results from sklearn as well as the original data. Do they agree ?

My implementation starts as follows:

```
def My_TreeRegressor(y,il,ir,max_depth,level=0):
    global yreg
    if (level==0): yreg=np.zeros(len(y))
```

Here, (somewhat clumsily) I keep track of the final tree regression in the global array `yreg`. Perhaps one of you can find a more elegant way of returning the final tree regression fit. Since we're only asking for the regression results in the discrete points x_p, `yreg` is simply an array in my implementation.

☛**Comment:** We're here using the decision tree as a regressor. From that perspective, when the x_p's are not equidistant it seems to me a somewhat crude approximation to take the leaf values to simply be

$$\frac{1}{N}\sum_{p=1}^{N} y_p$$

It would have made more sense to me to approximate them using a trapezoidal rule as

$$\frac{1}{(b-a)}\sum_{p=1}^{N-1} \frac{y_p + y_{p+1}}{2}(x_{p+1} - x_p)$$

With the 2 expressions being equal for equidistant points. Since Scikit-learn is mainly aimed at classification the sklearn regressor does not seem to make this distinction as far as I can tell.