

****REGULARIZATION****

ABOUT THE FASHION MNIST DATASET

Fashion-MNIST is a dataset of Zalando's article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255. The training and test data sets have 785 columns. The first column consists of the class labels (see above), and represents the article of clothing. The rest of the columns contain the pixel-values of the associated image.

Labels

Each training and test example is assigned to one of the following labels:

- 0 T-shirt/top
- 1 Trouser
- 2 Pullover
- 3 Dress
- 4 Coat
- 5 Sandal
- 6 Shirt
- 7 Sneaker
- 8 Bag
- 9 Ankle boot



TOPICS IN THIS ASSIGNMENT

- Importing and understanding dataset
- EDA
- Preparing data
- Building the model
- Compiling and fitting the model
- Prediction on test data
- Evaluating the model
- Regularization techniques
- Rebuilding the model using dropout
- Rebuilding the model using l2
- Rebuilding the model using l1
- Rebuilding the model using earlystopping

How To Load Dataset?

- Documentation Link - https://www.tensorflow.org/api_docs/python/tf/keras/datasets
- Video link below

In []:

```
from IPython.display import YouTubeVideo
YouTubeVideo('Vm_wFo4j3So', width=600, height=300)
```

Out []:

1. Importing & Understanding Data

In []:

```
# Import fashion mnist from tensorflow.keras.datasets
```

```
# load the data using fashion_mnist.load_data and define train_images, train_labels, test_images, test_labels
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
32768/29515 [=====] - 0s 0us/step
40960/29515 [=====] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26427392/26421880 [=====] - 0s 0us/step
26435584/26421880 [=====] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
16384/5148 [=====] - 0s
0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [=====] - 0s 0us/step
4431872/4422102 [=====] - 0s 0us/step
```

In []:

```
# check the shape of train_images dataset
```

Out []:

```
(60000, 28, 28)
```

In []:

```
# check the shape of train_images single image (train_images[0])
```

Out []:

```
(28, 28)
```

In []:

```
# let's look at the first image which will show in the array form
```

Out []:

```
array([[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
```

```

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
0, 0, 13, 73, 0, 0, 1, 4, 0, 0, 0, 0, 1,
1, 0],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3,
0, 36, 136, 127, 62, 54, 0, 0, 0, 1, 3, 4, 0,
0, 3],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6,
0, 102, 204, 176, 134, 144, 123, 23, 0, 0, 0, 0, 12,
10, 0],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 155, 236, 207, 178, 107, 156, 161, 109, 64, 23, 77, 130,
72, 15],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
69, 207, 223, 218, 216, 216, 163, 127, 121, 122, 146, 141, 88,
172, 66],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0,
200, 232, 232, 233, 229, 223, 223, 215, 213, 164, 127, 123, 196,
229, 0],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
183, 225, 216, 223, 228, 235, 227, 224, 222, 224, 221, 223, 245,
173, 0],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
193, 228, 218, 213, 198, 180, 212, 210, 211, 213, 223, 220, 243,
202, 0],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 3, 0, 12,
219, 220, 212, 218, 192, 169, 227, 208, 218, 224, 212, 226, 197,
209, 52],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 0, 99,
244, 222, 220, 218, 203, 198, 221, 215, 213, 222, 220, 245, 119,
167, 56],
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 55,
236, 228, 230, 228, 240, 232, 213, 218, 223, 234, 217, 217, 209,
92, 0],
[ 0, 0, 1, 4, 6, 7, 2, 0, 0, 0, 0, 0, 0, 237,
226, 217, 223, 222, 219, 222, 221, 216, 223, 229, 215, 218, 255,
77, 0],
[ 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 62, 145, 204, 228,
207, 213, 221, 218, 208, 211, 218, 224, 223, 219, 215, 224, 244,
159, 0],
[ 0, 0, 0, 0, 18, 44, 82, 107, 189, 228, 220, 222, 217,
226, 200, 205, 211, 230, 224, 234, 176, 188, 250, 248, 233, 238,
215, 0],
[ 0, 57, 187, 208, 224, 221, 224, 208, 204, 214, 208, 209, 200,
159, 245, 193, 206, 223, 255, 255, 221, 234, 221, 211, 220, 232,
246, 0],
[ 3, 202, 228, 224, 221, 211, 211, 214, 205, 205, 205, 220, 240,
80, 150, 255, 229, 221, 188, 154, 191, 210, 204, 209, 222, 228,
225, 0],
[ 98, 233, 198, 210, 222, 229, 229, 234, 249, 220, 194, 215, 217,
241, 65, 73, 106, 117, 168, 219, 221, 215, 217, 223, 223, 224,
229, 29],
[ 75, 204, 212, 204, 193, 205, 211, 225, 216, 185, 197, 206, 198,
213, 240, 195, 227, 245, 239, 223, 218, 212, 209, 222, 220, 221,
230, 67],
[ 48, 203, 183, 194, 213, 197, 185, 190, 194, 192, 202, 214, 219,
221, 220, 236, 225, 216, 199, 206, 186, 181, 177, 172, 181, 205,
206, 115],
[ 0, 122, 219, 193, 179, 171, 183, 196, 204, 210, 213, 207, 211,
210, 200, 196, 194, 191, 195, 191, 198, 192, 176, 156, 167, 177,
210, 92],
[ 0, 0, 74, 189, 212, 191, 175, 172, 175, 181, 185, 188, 189,
188, 193, 198, 204, 209, 210, 210, 211, 188, 188, 194, 192, 216,
170, 0],
[ 2, 0, 0, 0, 66, 200, 222, 237, 239, 242, 246, 243, 244,
221, 220, 193, 191, 179, 182, 182, 181, 176, 166, 168, 99, 58,
0, 0],
[ 0, 0, 0, 0, 0, 0, 0, 40, 61, 44, 72, 41, 35,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0],

```

```
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0],
 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0]], dtype=uint8)
```

Observation from the array output

- You can see value in the array ranging from 0-255 depicting RGB color.

```
In []:

# check length of train_labels

Out[:]:

60000

In []:

# check the labels of train data

Out[:]:

array([9, 0, 0, ..., 3, 0, 5], dtype=uint8)

In []:

# check shape of test data

Out[:]:

(10000, 28, 28)

In []:

# check the labels of test data

Out[:]:

array([9, 2, 1, ..., 8, 1, 5], dtype=uint8)
```

How to plot an image?

- You can find a simple way to plot an image of MNIST dataset by watching the video below.

```
In []:

YouTubeVideo('2JhLogJAUGc', width=600, height=300)

Out[:]:
```

2. EDA

How to plot multiple images of the fashion MNIST?

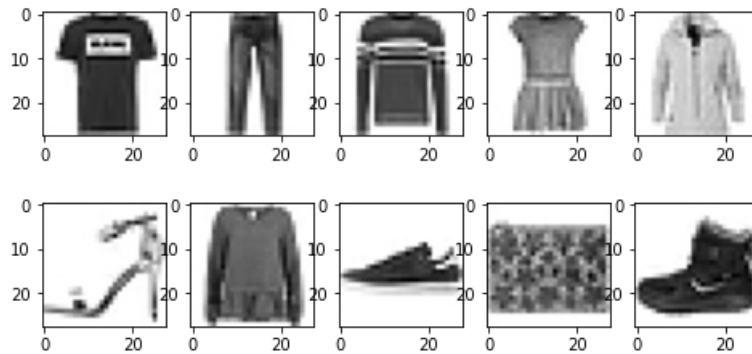
- Follow the comments below in the code to plot.

```
In []:

# importing pyplot and numpy for plotting images of 0-9
```

```
# defining subplots (2,5)
fig, ax = # code here

# looping over ax.flatten(), and plotting each digit
for i, ax in enumerate(ax.flatten()):
    # choosing each digit occuring at its first instance using np.argmax
    im_idx = # code here
    # reshaping the selected digit to (28, 28) from (1, 28, 28)
    plottable_image = # code here
    # now pass this plottable_image to ax.imshow
    ax.imshow(plottable_image, cmap='gray_r')
```



3. Preparing the data

How to prepare the image for model building?

- Follow the comments below to understand the process.

In []:

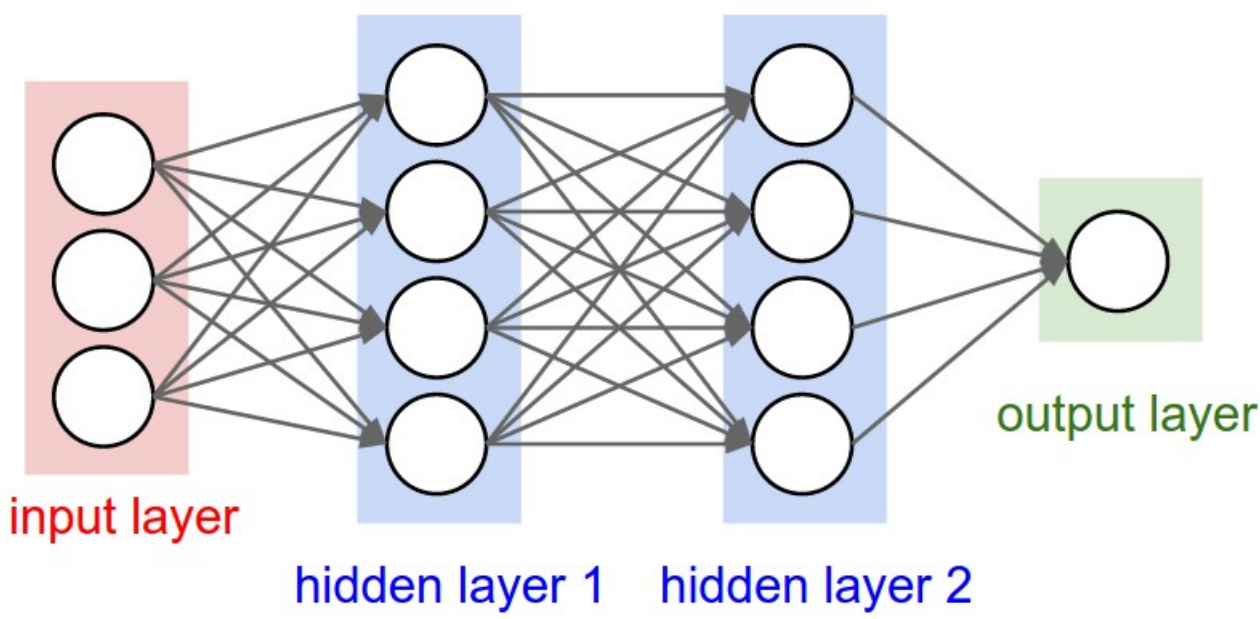
```
# reshape train_images from (60000, 28, 28) to (60000, 28*28)

# convert dtype of train_images from uint8 to float32

# reshape train_images from (10000, 28, 28) to (10000, 28*28)

# convert dtype of test_images from uint8 to float32
```

4.Model 1: Building the model



How To Build Model?

- Documentation Link - https://www.tensorflow.org/api_docs/python/tf/keras/Sequential
- Watch the video below & follow the steps in code cells later.

In []:

```
YouTubeVideo('FK77zZxaBoI', width=600, height=300)
```

Out[]:

In []:

```
YouTubeVideo('VGCHcgmZu24', width=600, height=300)
```

Out[]:

Let's import necessary libraries and define our model by following the comments

In []:

```
# importing keras and layers from tensorflow
```

```
# define the model and its network architecture  
# define three dense layers having first two layers with 512 neurons & activation='relu'  
# define third layer with 10 neurons & activation = 'softmax'  
model = # code here
```

5. Compiling & fitting the model

In the next cell, we will compile our model.

In []:

```
# compile the model with optimizer='rmsprop', loss='sparse_categorical_crossentropy', & metrics=['accuracy']
```

Then we will fit our model

In []:

```
# fit the model with epochs=10, batch_size=128

Epoch 1/10
469/469 [=====] - 9s 17ms/step - loss: 0.5396 - accuracy: 0.8048
Epoch 2/10
469/469 [=====] - 8s 17ms/step - loss: 0.3757 - accuracy: 0.8617
Epoch 3/10
469/469 [=====] - 8s 17ms/step - loss: 0.3316 - accuracy: 0.8776
Epoch 4/10
469/469 [=====] - 8s 17ms/step - loss: 0.3109 - accuracy: 0.8846
Epoch 5/10
469/469 [=====] - 10s 22ms/step - loss: 0.2946 - accuracy: 0.8915
Epoch 6/10
469/469 [=====] - 8s 16ms/step - loss: 0.2796 - accuracy: 0.8972
Epoch 7/10
469/469 [=====] - 8s 17ms/step - loss: 0.2677 - accuracy: 0.9022
Epoch 8/10
469/469 [=====] - 8s 17ms/step - loss: 0.2584 - accuracy: 0.9044
Epoch 9/10
469/469 [=====] - 8s 17ms/step - loss: 0.2496 - accuracy: 0.9057
Epoch 10/10
469/469 [=====] - 8s 16ms/step - loss: 0.2449 - accuracy: 0.9101

<keras.callbacks.History at 0x7fa7cdcf2d10>
```

Out []:

6. Prediction on test data

In the next cell, we will take first 10 images of test data

In []:

```
# define a variable test_digits and store the first 10 images of test data
```

Then we will predict on those 10 images

In []:

```
# predict the test_digits using our model
```

We will check prediction on first image in next cell

In []:

```
# check the first image prediction from predictions
```

Out []:

```
array([8.8619967e-10, 1.0212713e-12, 5.0321773e-09, 1.4206853e-09,
       4.9355240e-09, 5.2465132e-04, 7.5050941e-08, 9.8364393e-04,
       1.7612087e-09, 9.9849164e-01], dtype=float32)
```

Previous output shows probability of first image being either one of 10 products. For example, probability of first image being 0 is 8.8619967e-10 which is very very low.

In []:

```
# checking the index having maximum prediction
```

Out []:

9

In next cell, we can see the maximum prediction for first image is 0.99 at index 9..

In []:

```
# checking the index value having maximum prediction
```

Out []:

0.99849164

In next cell, we are confirming whether our prediction is right or wrong by checking label.

In []:

```
# checking the label for that index having maximum prediction
```

Out []:

9

7. Evaluating the model

Now we evaluate our model on unseen data, which is test set.

In []:

```
# check the loss and accuracy for test data using model.evaluate
test_loss, test_acc = # code here
```

```
313/313 [=====] - 1s 4ms/step - loss: 0.4655 - accuracy: 0.8611
```

In []:

```
# print test accuracy
print(f"test_acc: {test_acc}")
```

```
test_acc: 0.8611000180244446
```

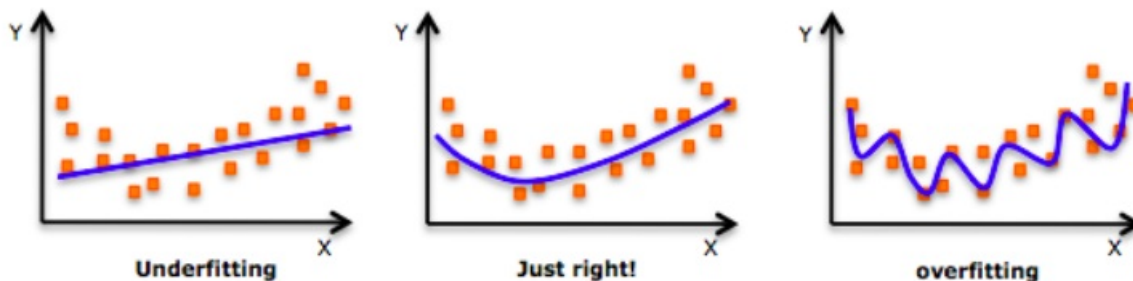
Observation from evaluation

- We can see that our train accuracy is 91.01% and our test accuracy is 86.11% which clearly shows the case of overfitting.

REGULARIZATION TECHNIQUES

What is Regularization?: It is a technique which makes slight modifications to the learning algorithm such that the model generalizes better. It is a way to overcome overfitting problem in machine learning/deep learning problem. This in turn improves the model's performance on the unseen data as well.

Overfitting: It is a situation where your model performed exceptionally well on train data but was not able to predict test data.



How do we deal with regularization?

There are multiple ways to deal with regularization

- 1. Dropouts
- 2. L2 & L1 regularization
- 3. Early stopping
- 4. Data augmentation
- 5. Tweaking the architecture

We have covered 5th tweaking the architecture method for regularization in our previous assignment of "Training Neural Networks". Data augmentation (4th) method will be covered in a separate assignment in computer vision.


Here we will cover first 3 ways mentioned above to show you how you can avoid overfitting.

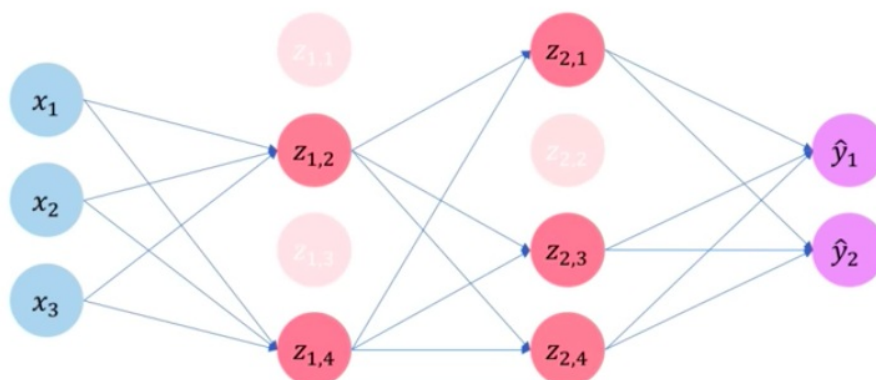
Dropouts

Dropout technique works by randomly reducing the number of interconnecting neurons within a neural network. At every training step, each neuron has a chance of being left out, or rather, dropped out of the collated contribution from connected neurons. This technique minimizes overfitting because each neuron becomes independently sufficient, in the sense that the neurons within the layers learn weight values that are not based on the cooperation of its neighbouring neurons.

Regularization I: Dropout

- During training, randomly set some activations to 0
 - Typically 'drop' 50% of activations in layer
 - Forces network to not rely on any 1 node

 `tf.nn.dropout(hiddenLayer, p=0.5)`



Dropout in implementation

Check the documentation of dropout - https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout

```
• from tensorflow.keras.layers import Dropout #importing the dropout layer
• model = Sequential([
•     Dense(units = 128, activation="relu", input_shape=
•         (train_data.shape[1],)),
•     Dropout(0.2),
•     Dense(units = 128, activation="relu"),
•     Dropout(0.2),
•     Dense(units = 128, activation="relu"),
•     Dropout(0.2),
•     Dense(units = 128, activation="relu"),
•     Dropout(0.2),
•     Dense(units = 128, activation="relu"),
```

In []:

```
# define the 2nd model and its network architecture
# define three dense layers having first two layers with 512 neurons & activation='relu'
# define third layer with 10 neurons & activation = 'softmax'
# add 2 dropout with value 0.3 (one each after first two dense layers)
from keras.layers.core import Dropout
model2 = # code here
```

In []:

```
# compile the model with optimizer='rmsprop', loss='sparse_categorical_crossentropy', & metrics=['accuracy']
```

In []:

```
# fit the model with epochs=10, batch_size=128
```

```
Epoch 1/10
469/469 [=====] - 9s 19ms/step - loss: 0.5770 - accuracy: 0.7894
Epoch 2/10
469/469 [=====] - 9s 19ms/step - loss: 0.4233 - accuracy: 0.8455
Epoch 3/10
469/469 [=====] - 9s 20ms/step - loss: 0.3900 - accuracy: 0.8598
Epoch 4/10
469/469 [=====] - 9s 20ms/step - loss: 0.3744 - accuracy: 0.8659
Epoch 5/10
469/469 [=====] - 9s 19ms/step - loss: 0.3619 - accuracy: 0.8703
Epoch 6/10
469/469 [=====] - 9s 19ms/step - loss: 0.3511 - accuracy: 0.8758
Epoch 7/10
469/469 [=====] - 9s 19ms/step - loss: 0.3469 - accuracy: 0.8783
Epoch 8/10
469/469 [=====] - 9s 19ms/step - loss: 0.3412 - accuracy: 0.8787
Epoch 9/10
469/469 [=====] - 10s 20ms/step - loss: 0.3385 - accuracy: 0.8818
Epoch 10/10
469/469 [=====] - 9s 19ms/step - loss: 0.3348 - accuracy: 0.8832
```

Out[]:

In []:

```
<keras.callbacks.History at 0x7fa84b6cd510>
```

```
# check the loss and accuracy for test data using model.evaluate
```

```
313/313 [=====] - 1s 3ms/step - loss: 0.3642 - accuracy: 0.8822
```

Observation from evaluation


- We can see that our train accuracy is 88.32% and our test accuracy is 88.22% which clearly shows how well dropout dealt with overfitting.

L2 Regularization

L2 regularization forces weights toward zero but it does not make them exactly zero. L2 regularization acts like a force that removes a small percentage of weights at each iteration. Therefore, weights will never be equal to zero. There is an additional parameter to tune the L2 regularization term which is called regularization rate (lambda).

Note: Choosing an optimal value for lambda is important. If lambda is too high, the model becomes too simple and tends to underfit. On the other hand, if lambda is too low, the effect of regularization becomes negligible and the model is likely to overfit. If lambda is set to zero, then regularization will be completely removed (high risk of overfitting!).

$$= \operatorname{argmin}_{\beta \in \mathbb{R}^p} \underbrace{\|y - X\beta\|_2^2}_{\text{Loss}} + \lambda \underbrace{\|\beta\|_2^2}_{\text{Penalty}}$$

L2 Regularization 

L2 Regularization in implementation

Check L2 regularization documentation - https://www.tensorflow.org/api_docs/python/tf/keras/regularizers/L2

Listing 4.6 Adding L2 weight regularization to the model

```
from keras import regularizers

model = models.Sequential()
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
    activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
    activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

In []:

```
# import regularizers from keras
```

```
# define the 3rd model and its network architecture
# define three dense layers having first two layers with 512 neurons , activation='relu', & kernel_regul.
# define third layer with 10 neurons & activation = 'softmax'
```

```
model3 = # code here
```

In []:

```
# compile the model with optimizer='rmsprop', loss='sparse_categorical_crossentropy', & metrics=['accuracy']

model3.compile(optimizer="rmsprop",
               loss="sparse_categorical_crossentropy",
               metrics=["accuracy"])
```

In []:

```
# fit the model with epochs=10, batch_size=128
```

```
Epoch 1/10
469/469 [=====] - 10s 19ms/step - loss: 0.9587 - accuracy: 0.7923
Epoch 2/10
469/469 [=====] - 9s 19ms/step - loss: 0.5695 - accuracy: 0.8430
Epoch 3/10
469/469 [=====] - 9s 19ms/step - loss: 0.4914 - accuracy: 0.8555
Epoch 4/10
469/469 [=====] - 9s 19ms/step - loss: 0.4553 - accuracy: 0.8647
Epoch 5/10
469/469 [=====] - 9s 19ms/step - loss: 0.4339 - accuracy: 0.8693
Epoch 6/10
469/469 [=====] - 9s 19ms/step - loss: 0.4173 - accuracy: 0.8733
Epoch 7/10
469/469 [=====] - 9s 19ms/step - loss: 0.4068 - accuracy: 0.8759
Epoch 8/10
469/469 [=====] - 9s 19ms/step - loss: 0.3955 - accuracy: 0.8781
Epoch 9/10
469/469 [=====] - 9s 20ms/step - loss: 0.3892 - accuracy: 0.8801
Epoch 10/10
469/469 [=====] - 9s 19ms/step - loss: 0.3817 - accuracy: 0.8825
```

Out []:

```
<keras.callbacks.History at 0x7f735f855a50>
```

In []:

```
# check the loss and accuracy for test data using model.evaluate
```

```
313/313 [=====] - 2s 4ms/step - loss: 0.4167 - accuracy: 0.8733
```

Observation from evaluation

- We can see that our train accuracy is 88.25% and our test accuracy is 87.33% which shows how well L2 regularization worked with overfitting but it's not better than dropout. We can still see some overfitting.

L1 Regularization

It adds an L1 penalty that is equal to the absolute value of the magnitude of coefficient, or simply restricting the size of coefficients.

L1 Regularization

$$\text{Cost} = \sum_{i=0}^N (y_i - \sum_{j=0}^M x_{ij} W_j)^2 + \lambda \sum_{j=0}^M |W_j|$$

L2 Regularization

$$\text{Cost} = \underbrace{\sum_{i=0}^N (y_i - \sum_{j=0}^M x_{ij} W_j)^2}_{\text{Loss function}} + \underbrace{\lambda \sum_{j=0}^M W_j^2}_{\text{Regularization Term}}$$

Loss function

Regularization
Term

L1 regularization in implementation

Check L1 regularization documentation -https://www.tensorflow.org/api_docs/python/tf/keras/regularizers/L1

```
from keras.regularizers import l1
from keras import backend as K
K.clear_session()
model = Sequential()
model.add(Dense(1000, input_dim=5, kernel_initializer='normal', kernel_regularizer=l1(0.001), activation='relu'))
model.add(Dense(500, kernel_initializer='normal', kernel_regularizer=l1(0.001), activation='relu'))
model.add(Dense(100, kernel_initializer='normal', kernel_regularizer=l1(0.001), activation='relu'))

model.add(Dense(2, activation='softmax'))
tensorboard = TensorBoard(log_dir = "/content/datalab/logs/tensor_22", histogram_freq=1000, batch_size=5000)

adam = Adam(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=adam, metrics=['accuracy'])
```

In []:

```
# define the 4th model and its network architecture
# define three dense layers having first two layers with 512 neurons , activation='relu', & kernel_regularizer=l1(0.001)
# define third layer with 10 neurons & activation = 'softmax'
```

In []:

```
# compile the model with optimizer='rmsprop', loss='sparse_categorical_crossentropy', & metrics=['accuracy']
```

In []:

```
# fit the model with epochs=10, batch_size=128
```

```
Epoch 1/10
469/469 [=====] - 10s 20ms/step - loss: 2.9857 - accuracy: 0.7411
Epoch 2/10
469/469 [=====] - 9s 19ms/step - loss: 1.0575 - accuracy: 0.8026
Epoch 3/10
469/469 [=====] - 11s 22ms/step - loss: 0.9609 - accuracy: 0.8181
Epoch 4/10
469/469 [=====] - 9s 19ms/step - loss: 0.9164 - accuracy: 0.8260
Epoch 5/10
469/469 [=====] - 9s 19ms/step - loss: 0.8891 - accuracy: 0.8313
Epoch 6/10
469/469 [=====] - 9s 19ms/step - loss: 0.8743 - accuracy: 0.8342
Epoch 7/10
469/469 [=====] - 9s 20ms/step - loss: 0.8603 - accuracy: 0.8386
Epoch 8/10
469/469 [=====] - 9s 20ms/step - loss: 0.8522 - accuracy: 0.8399
Epoch 9/10
469/469 [=====] - 9s 20ms/step - loss: 0.8450 - accuracy: 0.8413
Epoch 10/10
469/469 [=====] - 9s 20ms/step - loss: 0.8392 - accuracy: 0.8434
```

Out []:

```
<keras.callbacks.History at 0x7f735f646f90>
```

In []:

```
# check the loss and accuracy for test data using model.evaluate
```

```
313/313 [=====] - 2s 4ms/step - loss: 0.8673 - accuracy: 0.8308
```

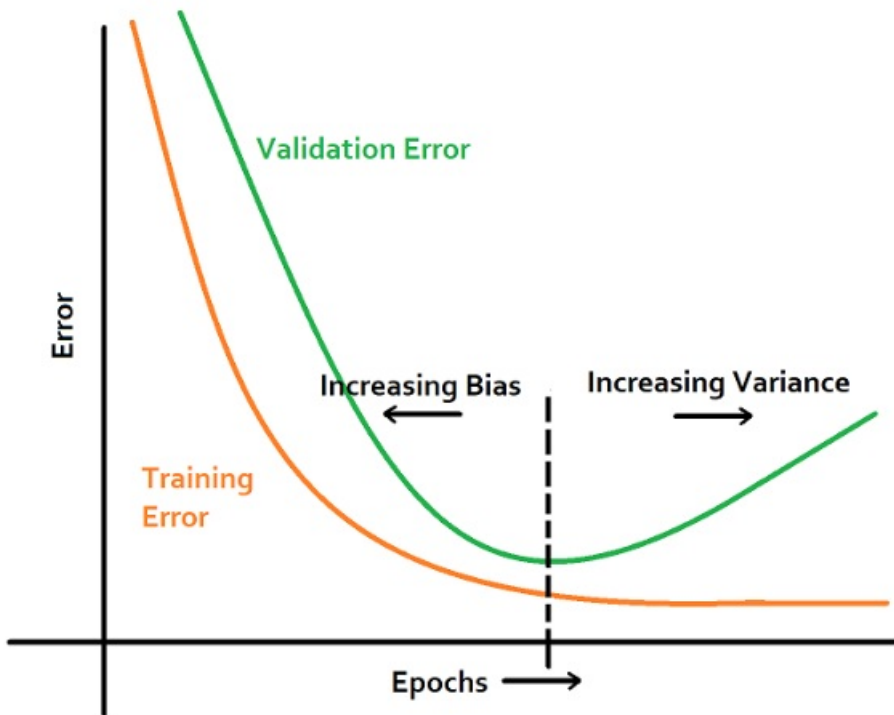
Observation from evaluation

- We can see that our train accuracy is 84.34% and our test accuracy is 83.08% which shows how our training accuracy dropped from 88.32% to 84.3% and our test accuracy dropped from 88.22% to 83.08% as compared to our 2nd model. It does reduce overfitting as compared to our first model, but it did not perform well as compared to our 2nd model and 3rd model.

Early Stopping

What is early stopping? - Early stopping is a method that allows you to specify an arbitrary large number of training epochs and stop training once the model performance stops improving on a hold out validation dataset.

Why we use early stopping? - Too many epochs can lead to overfitting of the training dataset, whereas too few may result in an underfit model.



Early stopping in implementation

early stopping documentation link - https://keras.io/api/callbacks/early_stopping/

Code Editor

```
from keras.callbacks import EarlyStopping
[other code...]
my_callbacks = [EarlyStopping(patience=10, monitor="accuracy")]
model.fit(X_train, y_train, validation_data = (X_val, y_val), epochs=100, callback
```

In []:

```
# import tensorflow as tf
import tensorflow as tf

# define callbacks variable and import earlystopping with monitor='loss', patience=2
callbacks = # code here

# define the 5th model and its network architecture
# define three dense layers having first two layers with 512 neurons & activation='relu'
# define third layer with 10 neurons & activation = 'softmax'
```

In []:

```
# compile the model with optimizer='rmsprop', loss='sparse_categorical_crossentropy', & metrics=['accuracy']
```

In []:

```
# fit the model with epochs=10, batch_size=128, callbacks=[callbacks], validation_data = (test_images, test_labels)
```

```

Epoch 1/10
469/469 [=====] - 9s 18ms/step - loss: 0.5391 - accuracy: 0.8027 - val_loss: 0.4102 - val_accuracy: 0.8521
Epoch 2/10
469/469 [=====] - 8s 18ms/step - loss: 0.3752 - accuracy: 0.8605 - val_loss: 0.3979 - val_accuracy: 0.8530
Epoch 3/10
469/469 [=====] - 8s 18ms/step - loss: 0.3361 - accuracy: 0.8760 - val_loss: 0.4798 - val_accuracy: 0.8472
Epoch 4/10
469/469 [=====] - 8s 18ms/step - loss: 0.3093 - accuracy: 0.8849 - val_loss: 0.3915 - val_accuracy: 0.8613
Epoch 5/10
469/469 [=====] - 8s 18ms/step - loss: 0.2899 - accuracy: 0.8929 - val_loss: 0.3766 - val_accuracy: 0.8774
Epoch 6/10
469/469 [=====] - 8s 18ms/step - loss: 0.2773 - accuracy: 0.8985 - val_loss: 0.4837 - val_accuracy: 0.8568
Epoch 7/10
469/469 [=====] - 8s 18ms/step - loss: 0.2658 - accuracy: 0.9019 - val_loss: 0.3920 - val_accuracy: 0.8716
Epoch 8/10
469/469 [=====] - 8s 18ms/step - loss: 0.2580 - accuracy: 0.9046 - val_loss: 0.3711 - val_accuracy: 0.8852
Epoch 9/10
469/469 [=====] - 8s 18ms/step - loss: 0.2491 - accuracy: 0.9079 - val_loss: 0.3971 - val_accuracy: 0.8705
Epoch 10/10
469/469 [=====] - 9s 18ms/step - loss: 0.2404 - accuracy: 0.9122 - val_loss: 0.4073 - val_accuracy: 0.8843

```

Out[]:

<keras.callbacks.History at 0x7f7358658bd0>

Observation from evaluation

- We can see that our train accuracy is 91.22% and our test accuracy is 88.43% which shows how well early stopping method worked with overfitting but it's not better than dropout. We can still see some overfitting. However our test accuracy is better than dropout test accuracy. In terms of accuracy it is better but in terms of generalization, our dropout model which is 2nd model is best.

Summary

In this assignment, we learnt how we can implement multiple methods to deal with overfitting. We learnt the implementation of dropout, l2, l1, early stopping methods of regularization.

Now you can play and tweak the parameters to get more insights from these methods.



FEEDBACK FORM

Please fill the form and help us to improve

<https://forms.zohopublic.in/cloudyml/form/CloudyMLDeepLearningFeedbackForm/formperma/VCFbldnXAnbcgAll0IWv2blgHdSldheO4RfktM>

