In this Lab Assignment I've focused on the Mnist dataset (used in point 1, 2.1 and 2.2), which is a built in dataset from keras. After that I thought it was interesting to compare the performance with more complex data (2.3 and 2.4) and a more complex model (2.5).

Figura 2 - Mnist Dataset

Figura 2 - Fashion Mnist Dataset

1. ***Tensorflow:***

```
(x_train, y_train), (x_test, y_test) = load_data()

#Create Neural Network
net = NeuralNetwork([784,512,256,10])
#Display Number of Parameters
net.info()

#Hyperparameters
batch_size = 32
epochs = 11
steps_per_epoch = int(x_train.shape[0]/batch_size)
lr = 0.005
print('Steps per epoch', steps_per_epoch)

#Trainning and Validation
history = net.train(
    x_train,y_train,
    x_test, y_test,
    epochs, steps_per_epoch,
    batch_size, lr)

#Graph
plot_results(history).show()
```

Started by selecting data using load_data() and split it into training and testing data:

```
def load_data():
    (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
    x_train = np.reshape(x_train, (x_train.shape[0], 784))/255.
    x_test = np.reshape(x_test, (x_test.shape[0], 784))/255.
    y_train = tf.keras.utils.to_categorical(y_train)
    y_test = tf.keras.utils.to_categorical(y_test)
    return (x_train, y_train), (x_test, y_test)
```

After that, initialized my NeuralNetwork with 1 input layer of 784 neurons (trying to cover 28x28 features), 2 hidden layers (one with 512 neurons and another with 256 neurons) and an output layer of 10 neurons (since there are 10 classes/different numbers to identify).

Inside the NeuralNetwork class the main functions are:

➔ Loss calculation, using softmax activation since the goal is to know if each example fits only one of the several classes. Softmax is the most used activation in this situations. I'm using the cross entropy variation for optimised numerical stability during training. For probabilities of each class should have used tf.nn.softmax instead.
This loss calculation expect the linear part of neurons without the non-linear activation function. For that, last layer should be composed by 10 neurons, one for each class.

```
def compute_loss(self, A, Y):
    loss = tf.nn.softmax_cross_entropy_with_logits(Y,A)
    return tf.reduce_mean(loss)
```

➔ Predict, using foward propagation instead of backpropagation:

```
def predict(self, X):
    A = self.forward_pass(X)
    return tf.argmax(tf.nn.softmax(A), axis=1)
```

➔ Next, the main training mechanism, training on batch using gradient descent with automatic differentiation:

```
def train_on_batch(self, X, Y, lr):
    X = tf.convert_to_tensor(X, dtype=tf.float32)
    Y = tf.convert_to_tensor(Y, dtype=tf.float32)
    with tf.GradientTape(persistent=True) as tape:
        A = self.forward_pass(X)
        loss = self.compute_loss(A, Y)
    for i in range(1, self.L):
        self.dW[i] = tape.gradient(loss, self.W[i])
        self.db[i] = tape.gradient(loss, self.b[i])
    del tape
    self.update_params(lr)
    return loss.numpy()
```

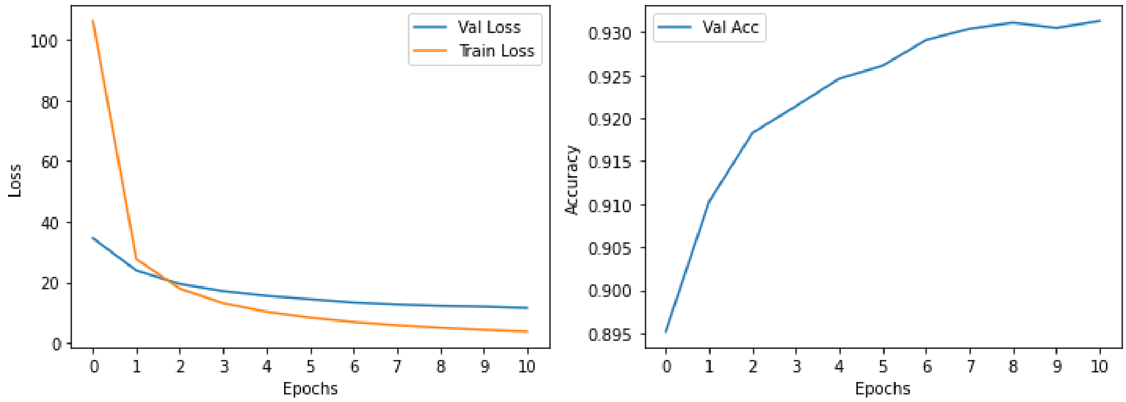
➔ And the "higher level" training, running all epochs, in each epoch run all the steps_per_epoch and in each step calculate the loss of the whole batch size.

```
def train(self, x_train, y_train, x_test, y_test, epochs, steps_per_epoch, batch_size, lr):
    history = {
        'val_loss':[],
        'train_loss':[],
        'val_acc':[]
    }
    for e in range(0, epochs):
        epoch_train_loss = 0.
        print('Epoch{}'.format(e), end='.')
        for i in range(0, steps_per_epoch):
            x_batch = x_train[i*batch_size:(i+1)*batch_size]
            y_batch = y_train[i*batch_size:(i+1)*batch_size]

            batch_loss = self.train_on_batch(x_batch, y_batch,lr)
            epoch_train_loss += batch_loss

            if i%int(steps_per_epoch/10) == 0:
                print(end='.')

        history['train_loss'].append(epoch_train_loss/steps_per_epoch)
        val_A = self.forward_pass(x_test)
        val_loss = self.compute_loss(val_A, y_test).numpy()
        #print(np.exp(val_loss)/np.exp(val_A))
        history['val_loss'].append(val_loss)
        val_preds = self.predict(x_test)
        val_acc =    np.mean(np.argmax(y_test, axis=1) == val_preds.numpy())
        history['val_acc'].append(val_acc)
        print('Val acc:',val_acc)
    return history
```

Before calling NeuralNetwork train function I defined:

- ➔ Epoch=11: With 11 I could take enough conclusions about the implementation, ideally this number should be around 50 but it would take too long
- ➔ Learning_Rate=0.005: Higher learning rate makes training less time consuming but might sacrífice some convergence at the end; while Lower learning rate would take too long to train
- ➔ Batch size=64: 32 It's the standard value on keras so I was going to use 32 but then I noticed 64 had significantly less oscillations. Higher batch size means faster training and better convergence, since true gradient is the mean loss over all batch points; Smaller batches improve generalization but takes much longer to train.

Training and Validation results were the following:



2. *Keras*

2.1. Mnist Sequential

This is the code, with comments:

```
class Keras_Sequential_Mnist:

    #Data Selection
    (x_train_full, y_train_full), (x_test, y_test) = mnist.load_data()

    X_valid, X_train = x_train_full[:5000] / 255.0, x_train_full[5000:] / 255.0
    y_valid, y_train = y_train_full[:5000], y_train_full[5000:]

    #Just to fix tensorflow and tensorflow-gpu conflict in my computer
    with tf.device('/CPU:0'):

        model = keras.models.Sequential()

    #Structure of the model
        model.add(Conv2D(64, (3, 3), activation='relu', input_shape=(28,28,1)))
        model.add(BatchNormalization())
        model.add(Conv2D(64, (3, 3), activation='relu'))
        model.add(BatchNormalization())
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Conv2D(64, (3, 3), activation='relu'))
        model.add(BatchNormalization())
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Flatten())
        model.add(Dense(512, activation='relu'))
        model.add(BatchNormalization())
        model.add(Dropout(0.5))
        model.add(Dense(10))
        model.add(Activation("softmax"))

    #Summary
        model.summary()

    #Image with model summary
        plot_model(model, to_file='model_Seq_Mnist.png', show_shapes=True)

    #Optimizer and compile
        INIT_LR = 0.01
        NUM_EPOCHS = 11
        opt = SGD(learning_rate=INIT_LR, momentum=0.9, decay=INIT_LR / NUM_EPOCHS)
        model.compile(loss="sparse_categorical_crossentropy", optimizer=opt,metrics=["accuracy"])

    #Trainning and Validation
        history = model.fit(X_train, y_train, epochs=NUM_EPOCHS, validation_data=(X_valid, y_valid))

    #Graph
        pd.DataFrame(history.history).plot(figsize=(8, 5))
        plt.grid(True)
        plt.gca().set_ylim(0, 1)
        plt.show()
```

Hyperparameters already explained in the previous model, only difference here is that I'm using a dynamic learning rate to improve convergence epoch after epoch and I'm using the standard batch size 32.

Regarding structure, I've used some 2D convolution layers with a 3 × 3 kernel and 64 filters as this kind of layers are used for achieving high accuracy in image recognition tasks. Didn't use padding and stride left as default. Bigger stride would simplify and reduce processing time, padding would help sizing the output channels to match the input channels of the next layer, no need here.

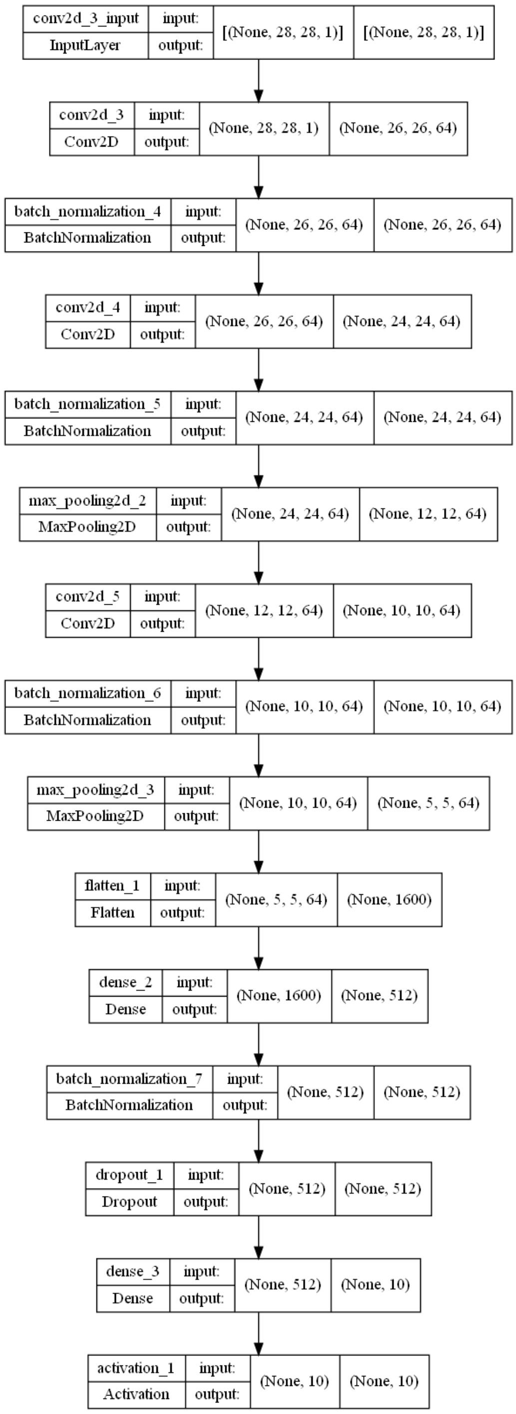
All hidden layers have ReLU activation and output layer has softmax activation.

ReLU is a very good activation for hidden layers because the output of ReLU does not have a maximum value, so it doesn't saturate like sigmoid activations for example. Another advantage is that ReLU is fast to compute.

MaxPooling helps with overfitting and it reduces the computational cost by reducing the number of parameters to learn and provides basic translation invariance to the internal representation. Used MaxPooling with pool size (2,2), to reduce size to half (horizontally and vertically)

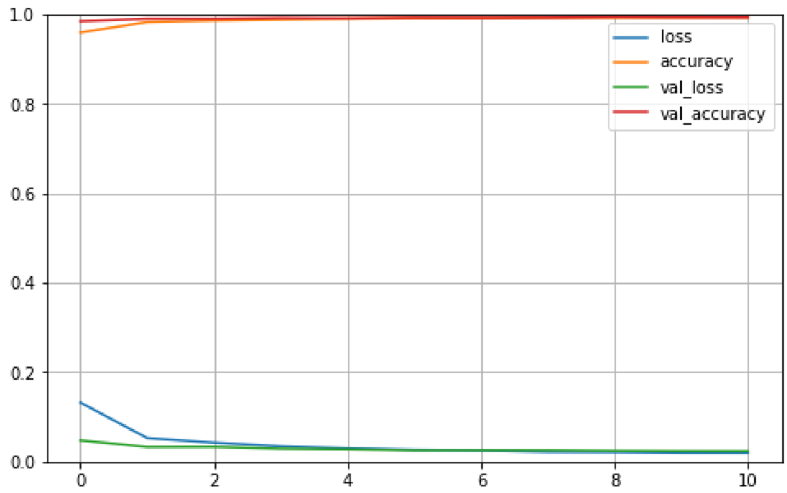
"Overusing" BatchNormalization, according to some tests, I've noticed around 2% increase on accuracy! It's a layer that normalizes its inputs preventing extreme values. Batch normalization applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1. Makes learning easier, improves accuracy and consequentially reduces number of epochs needed to train data to a certain accuracy value.

Flatten layer is needed to flatten the input into a single dimension, which is what the dense layer expects. Wich means that if it receives a input of 5x5x64, the output needs to be the multiplication of the 3 parameters, 1600.



Since the softmax activation has the same justification as the previous model, the last thing I'll talk about here is the Dropout layer, which is user for regularization and preventing overfitting. Dropout randomly sets an input neuron to 0 at each step of training to prevent model to adapt too much to the data (overfitting).

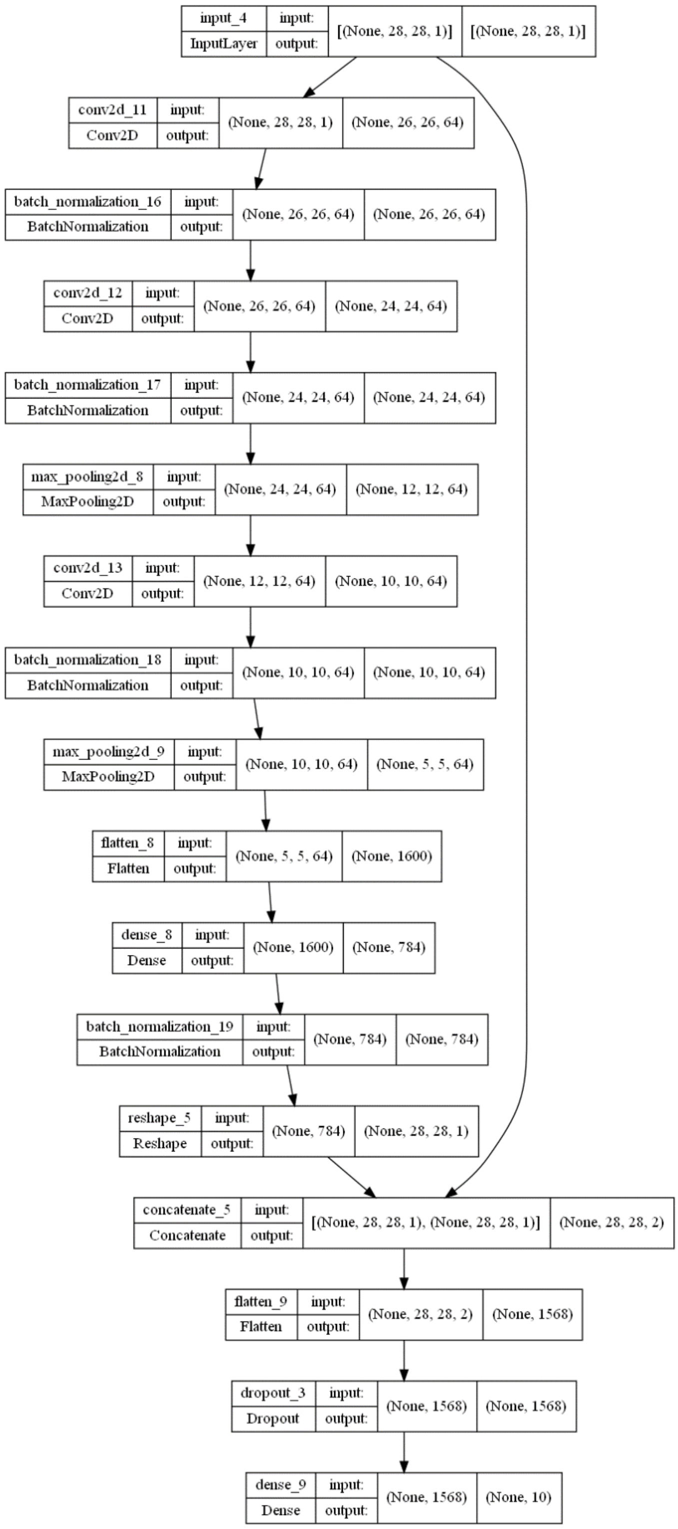
Training and Validation results were the following:



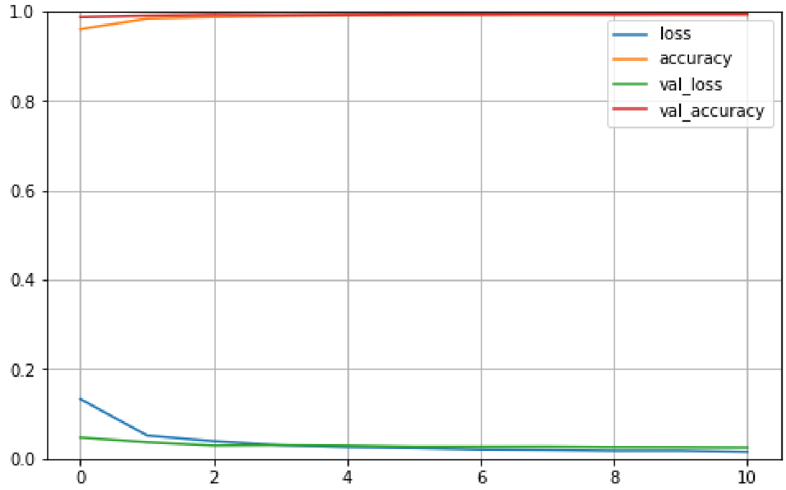
2.2. Mnist Functional

```
#Get Data
(x_train_full, y_train_full), (x_test, y_test) = mnist.load_data()

X_valid, X_train = x_train_full[:5000] / 255.0, x_train_full[5000:] / 255.0
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]

with tf.device('/CPU:0'):
    model = keras.models.Sequential()

    #Structure
    input_      =   Input(shape=[28, 28,1])
    hidden01    =   Conv2D(64,kernel_size=3, activation='relu') (input_)
    hidden_     =   BatchNormalization()(hidden01)
    hidden02    =   Conv2D(64,kernel_size=3, activation='relu') (hidden_)
    hidden_1    =   BatchNormalization()(hidden02)
    hidden03    =   MaxPooling2D(pool_size=(2))(hidden_1)
    hidden04    =   Conv2D(64,kernel_size=3, activation='relu') (hidden03)
    hidden_2    =   BatchNormalization()(hidden04)
    hidden05    =   MaxPooling2D(pool_size=(2))(hidden_2)
    flatten     =   Flatten(input_shape=[28, 28])(hidden05)
    hidden06    =   Dense((784), activation='relu')(flatten)
    hidden_3    =   BatchNormalization()(hidden06)
    reshap      =   Reshape((28, 28,1))(hidden_3)

    concat_     =   Concatenate()([input_, reshap])
    flatten2    =   Flatten(input_shape=[28, 28,1])(concat_)
    drop        =   Dropout(0.5)(flatten2)
    output      =   Dense(10, activation='softmax')(drop)

    model = keras.Model(inputs=[input_], outputs=[output] )

    #Summary
    model.summary()

    #Image with struct
    plot_model(model, to_file='model_Func_Mnist.png')

    #Optimizer
    INIT_LR = 0.01
    NUM_EPOCHS = 11
    opt = SGD(learning_rate=INIT_LR, momentum=0.9, decay=INIT_LR / NUM_EPOCHS)
    model.compile(loss="sparse_categorical_crossentropy", optimizer=opt,
                  metrics=["accuracy"])

    #Graph
    history = model.fit(X_train, y_train, epochs=NUM_EPOCHS, validation_data=(X_valid, y_valid))
    pd.DataFrame(history.history).plot(figsize=(8, 5))
    plt.grid(True)
    plt.gca().set_ylim(0, 1)
    plt.show()
```

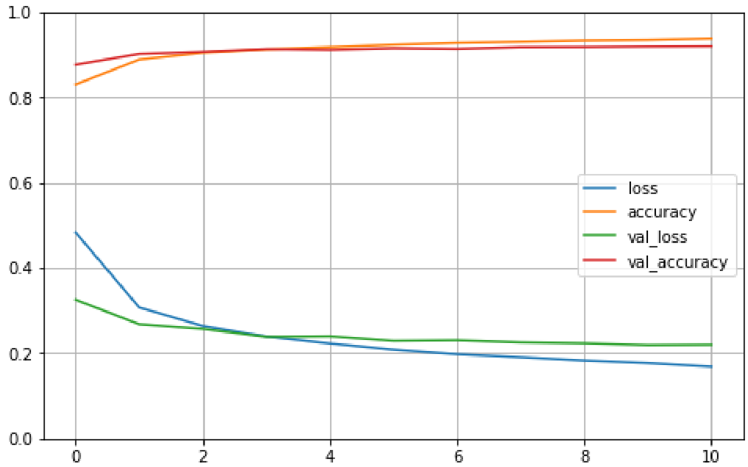
This architecture makes it possible for the neural network to learn both deep patterns and simple rules. One example is that functional allows to build hybrid models, with deep and wide characteristics.
There is little difference from this model to the previous (2.1), the only differences is the input connected to Dense layer before the output of the previous model, like in the following image:



The result was the same, the previous model was already fully optimized (99%).



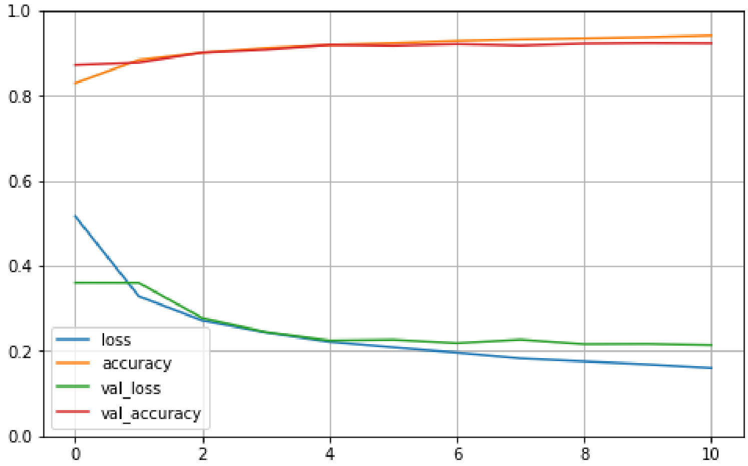
2.3. Fashion Sequential




```
Epoch 1/11
938/938 [==============================] - 208s 219ms/step - loss: 0.4837 - accuracy: 0.8305 - val_loss: 0.3257 - val_accuracy: 0.8771
Epoch 2/11
938/938 [==============================] - 201s 215ms/step - loss: 0.3076 - accuracy: 0.8890 - val_loss: 0.2681 - val_accuracy: 0.9021
Epoch 3/11
938/938 [==============================] - 189s 202ms/step - loss: 0.2641 - accuracy: 0.9049 - val_loss: 0.2574 - val_accuracy: 0.9070
Epoch 4/11
938/938 [==============================] - 196s 209ms/step - loss: 0.2394 - accuracy: 0.9127 - val_loss: 0.2386 - val_accuracy: 0.9136
Epoch 5/11
938/938 [==============================] - 200s 213ms/step - loss: 0.2230 - accuracy: 0.9183 - val_loss: 0.2398 - val_accuracy: 0.9118
Epoch 6/11
938/938 [==============================] - 192s 205ms/step - loss: 0.2086 - accuracy: 0.9240 - val_loss: 0.2293 - val_accuracy: 0.9155
Epoch 7/11
938/938 [==============================] - 191s 204ms/step - loss: 0.1977 - accuracy: 0.9283 - val_loss: 0.2306 - val_accuracy: 0.9141
Epoch 8/11
938/938 [==============================] - 189s 201ms/step - loss: 0.1906 - accuracy: 0.9309 - val_loss: 0.2257 - val_accuracy: 0.9181
Epoch 9/11
938/938 [==============================] - 190s 202ms/step - loss: 0.1829 - accuracy: 0.9338 - val_loss: 0.2235 - val_accuracy: 0.9185
Epoch 10/11
938/938 [==============================] - 193s 206ms/step - loss: 0.1771 - accuracy: 0.9352 - val_loss: 0.2193 - val_accuracy: 0.9197
Epoch 11/11
938/938 [==============================] - 189s 202ms/step - loss: 0.1689 - accuracy: 0.9384 - val_loss: 0.2200 - val_accuracy: 0.9204
```

Same model as 2.1 still performs really well, around 1-2% better than some models I found online.

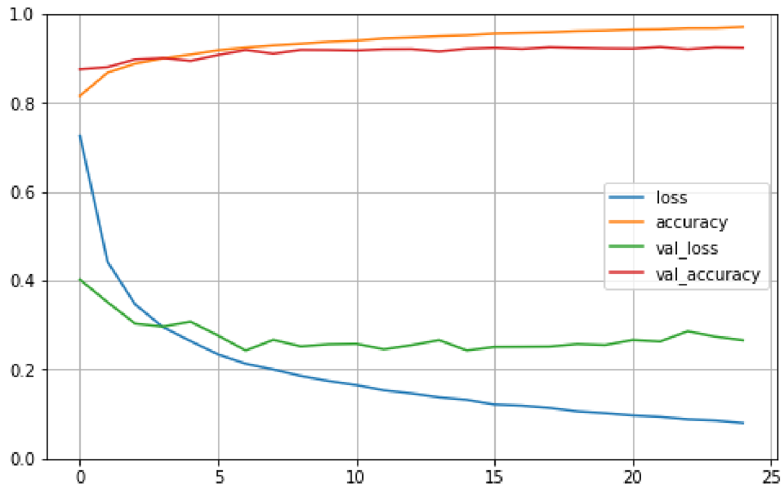
2.4. Fashion Mnist Functional




```
Epoch 1/11
938/938 [==============================] - 223s 236ms/step - loss: 0.5170 - accuracy: 0.8296 - val_loss: 0.3610 - val_accuracy: 0.8725
Epoch 2/11
938/938 [==============================] - 200s 213ms/step - loss: 0.3290 - accuracy: 0.8844 - val_loss: 0.3607 - val_accuracy: 0.8782
Epoch 3/11
938/938 [==============================] - 193s 206ms/step - loss: 0.2717 - accuracy: 0.9016 - val_loss: 0.2777 - val_accuracy: 0.9016
Epoch 4/11
938/938 [==============================] - 189s 202ms/step - loss: 0.2434 - accuracy: 0.9125 - val_loss: 0.2445 - val_accuracy: 0.9086
Epoch 5/11
938/938 [==============================] - 188s 200ms/step - loss: 0.2216 - accuracy: 0.9196 - val_loss: 0.2248 - val_accuracy: 0.9198
Epoch 6/11
938/938 [==============================] - 192s 204ms/step - loss: 0.2086 - accuracy: 0.9236 - val_loss: 0.2266 - val_accuracy: 0.9179
Epoch 7/11
938/938 [==============================] - 192s 205ms/step - loss: 0.1963 - accuracy: 0.9292 - val_loss: 0.2186 - val_accuracy: 0.9216
Epoch 8/11
938/938 [==============================] - 194s 206ms/step - loss: 0.1828 - accuracy: 0.9326 - val_loss: 0.2269 - val_accuracy: 0.9183
Epoch 9/11
938/938 [==============================] - 192s 204ms/step - loss: 0.1758 - accuracy: 0.9350 - val_loss: 0.2163 - val_accuracy: 0.9227
Epoch 10/11
938/938 [==============================] - 193s 206ms/step - loss: 0.1685 - accuracy: 0.9377 - val_loss: 0.2168 - val_accuracy: 0.9238
Epoch 11/11
938/938 [==============================] - 193s 206ms/step - loss: 0.1602 - accuracy: 0.9416 - val_loss: 0.2140 - val_accuracy: 0.9232
```

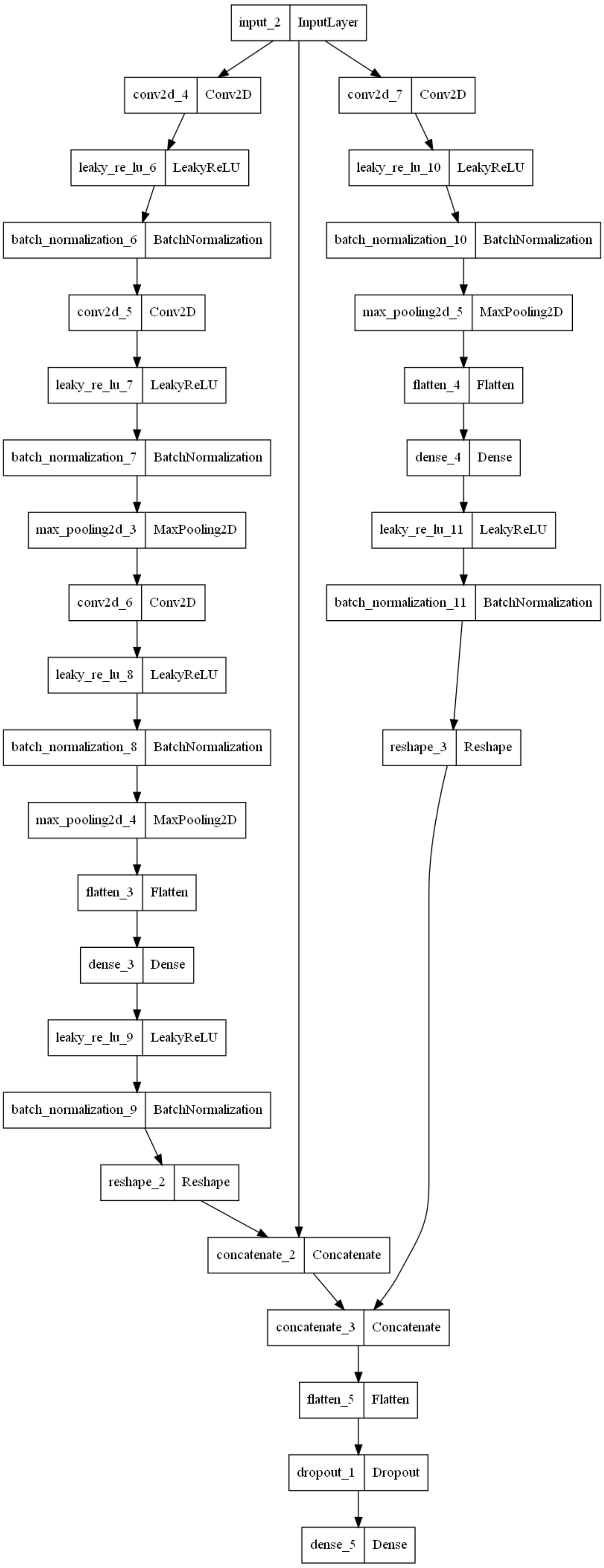
Same model as 2.2, looks the same as 2.3. Since accuracy and loss isn't constant everytime the code is executed I can't conclude that 2.4 is just slightly better than 2.3, but in this specific case, it seems that after epoch 5 the model 2.4 seems to perform just a little bit better.

2.5. Complex Fashion Mnist Functional



```
Epoch 2/25
938/938 [==============================] - 695s 741ms/step - loss: 0.4423 - accuracy: 0.8681 - val_loss: 0.3514 - val_accuracy: 0.8804
Epoch 3/25
938/938 [==============================] - 694s 740ms/step - loss: 0.3465 - accuracy: 0.8884 - val_loss: 0.3033 - val_accuracy: 0.8979
Epoch 4/25
938/938 [==============================] - 694s 740ms/step - loss: 0.2957 - accuracy: 0.9000 - val_loss: 0.2966 - val_accuracy: 0.9010
Epoch 5/25
938/938 [==============================] - 697s 743ms/step - loss: 0.2642 - accuracy: 0.9086 - val_loss: 0.3077 - val_accuracy: 0.8945
Epoch 6/25
938/938 [==============================] - 699s 745ms/step - loss: 0.2339 - accuracy: 0.9184 - val_loss: 0.2764 - val_accuracy: 0.9077
Epoch 7/25
938/938 [==============================] - 713s 760ms/step - loss: 0.2131 - accuracy: 0.9244 - val_loss: 0.2429 - val_accuracy: 0.9190
Epoch 8/25
938/938 [==============================] - 727s 775ms/step - loss: 0.2002 - accuracy: 0.9293 - val_loss: 0.2670 - val_accuracy: 0.9107
Epoch 9/25
938/938 [==============================] - 720s 768ms/step - loss: 0.1855 - accuracy: 0.9330 - val_loss: 0.2520 - val_accuracy: 0.9192
Epoch 10/25
938/938 [==============================] - 721s 768ms/step - loss: 0.1742 - accuracy: 0.9375 - val_loss: 0.2568 - val_accuracy: 0.9188
Epoch 11/25
938/938 [==============================] - 719s 767ms/step - loss: 0.1651 - accuracy: 0.9401 - val_loss: 0.2580 - val_accuracy: 0.9177
Epoch 12/25
938/938 [==============================] - 723s 770ms/step - loss: 0.1533 - accuracy: 0.9452 - val_loss: 0.2456 - val_accuracy: 0.9206
Epoch 13/25
938/938 [==============================] - 719s 767ms/step - loss: 0.1463 - accuracy: 0.9473 - val_loss: 0.2548 - val_accuracy: 0.9209
Epoch 14/25
938/938 [==============================] - 725s 773ms/step - loss: 0.1376 - accuracy: 0.9502 - val_loss: 0.2664 - val_accuracy: 0.9156
Epoch 15/25
938/938 [==============================] - 729s 777ms/step - loss: 0.1319 - accuracy: 0.9522 - val_loss: 0.2430 - val_accuracy: 0.9217
Epoch 16/25
938/938 [==============================] - 704s 750ms/step - loss: 0.1213 - accuracy: 0.9560 - val_loss: 0.2509 - val_accuracy: 0.9239
Epoch 17/25
938/938 [==============================] - 700s 746ms/step - loss: 0.1187 - accuracy: 0.9574 - val_loss: 0.2512 - val_accuracy: 0.9212
Epoch 18/25
938/938 [==============================] - 703s 749ms/step - loss: 0.1137 - accuracy: 0.9589 - val_loss: 0.2516 - val_accuracy: 0.9252
Epoch 19/25
938/938 [==============================] - 716s 763ms/step - loss: 0.1058 - accuracy: 0.9614 - val_loss: 0.2574 - val_accuracy: 0.9237
Epoch 20/25
938/938 [==============================] - 708s 755ms/step - loss: 0.1017 - accuracy: 0.9626 - val_loss: 0.2551 - val_accuracy: 0.9225
Epoch 21/25
938/938 [==============================] - 711s 758ms/step - loss: 0.0968 - accuracy: 0.9649 - val_loss: 0.2666 - val_accuracy: 0.9220
Epoch 22/25
938/938 [==============================] - 711s 758ms/step - loss: 0.0935 - accuracy: 0.9656 - val_loss: 0.2632 - val_accuracy: 0.9257
Epoch 23/25
938/938 [==============================] - 717s 764ms/step - loss: 0.0879 - accuracy: 0.9680 - val_loss: 0.2862 - val_accuracy: 0.9206
Epoch 24/25
938/938 [==============================] - 738s 787ms/step - loss: 0.0853 - accuracy: 0.9681 - val_loss: 0.2740 - val_accuracy: 0.9249
Epoch 25/25
938/938 [==============================] - 750s 800ms/step - loss: 0.0797 - accuracy: 0.9709 - val_loss: 0.2660 - val_accuracy: 0.9240
```

Model is in the next page. But the conclusion is that in this last model the overfitting (gap between val_loss and loss) at epoch 10 is much more noticeable than in 2.3 and 2.4.



All this 6 models are displayed as classes in tutorial.py. When the code is finished the 6 plots will be displayed on Spyder if the right utilities are installed (I needed to install graphviz) and the last 5 (all except the 1st, only using tensorflow) models summary with be created as .png file on the tutorial.py folder.