CO543 LAB 04 PART 2

DEVINDI G.A.I

E/17/058

In [1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import random
# TensorFlow
import tensorflow as tf

directory = "/content/drive/MyDrive/Colab Notebooks/CO543/Lab04/"
```

Helper functions to display the images with predicted labels and the predicted probabilities

In [2]:
```
def plot_image(i, predictions_array, true_label, img):
  true_label, img = true_label[i], img[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])

  plt.imshow(img, cmap=plt.cm.binary)

  predicted_label = np.argmax(predictions_array)
  if predicted_label == true_label:
    color = 'blue'
  else:
    color = 'red'

  plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
                                100*np.max(predictions_array),
                                class_names[true_label]),
                                color=color)

def plot_value_array(i, predictions_array, true_label):
  true_label = true_label[i]
  plt.grid(False)
  plt.xticks(range(10))
  plt.yticks([])
  thisplot = plt.bar(range(10), predictions_array, color="#777777")
  plt.ylim([0, 1])
  predicted_label = np.argmax(predictions_array)

  thisplot[predicted_label].set_color('red')
  thisplot[true_label].set_color('blue')
```

TASK 1

Image Acquisition

In [3]:
```
#read the training dataset
train_data = pd.read_csv(directory + 'fashion-mnist_train.csv')
train_data.head()
```

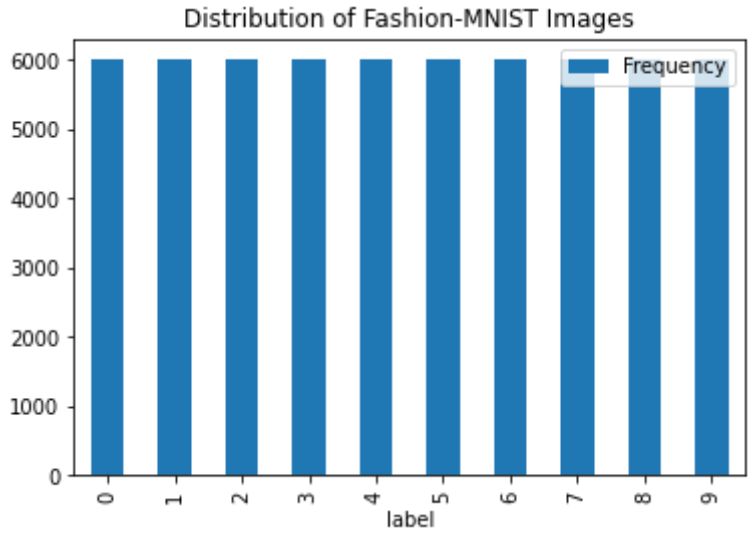
Out[3]:

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel78
0	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	
1	9	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	
2	6	0	0	0	0	0	0	0	5	0	...	0	0	0	30	43	0	
3	0	0	0	0	1	2	0	0	0	0	...	3	0	0	0	0	1	
4	3	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	

5 rows × 785 columns

In [4]:
```
#display the number of occurences of each class

train_data.groupby('label').count()[['pixel1']].rename(columns={'pixel1': 'Frequency'}).plot(kind='bar')

plt.title("Distribution of Fashion-MNIST Images")
```

Out[4]: Text(0.5, 1.0, 'Distribution of Fashion-MNIST Images')



Each class has an equaly distributed number of images

In [5]:
```
#read the test data set
test_data = pd.read_csv(directory + 'fashion-mnist_test.csv')
test_data.head()
```

Out[5]:

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel78
0	0	0	0	0	0	0	0	0	9	8	...	103	87	56	0	0	0	
1	1	0	0	0	0	0	0	0	0	0	...	34	0	0	0	0	0	
2	2	0	0	0	0	0	0	14	53	99	...	0	0	0	0	63	53	3
3	2	0	0	0	0	0	0	0	0	0	...	137	126	140	0	133	224	22
4	3	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	

5 rows × 785 columns

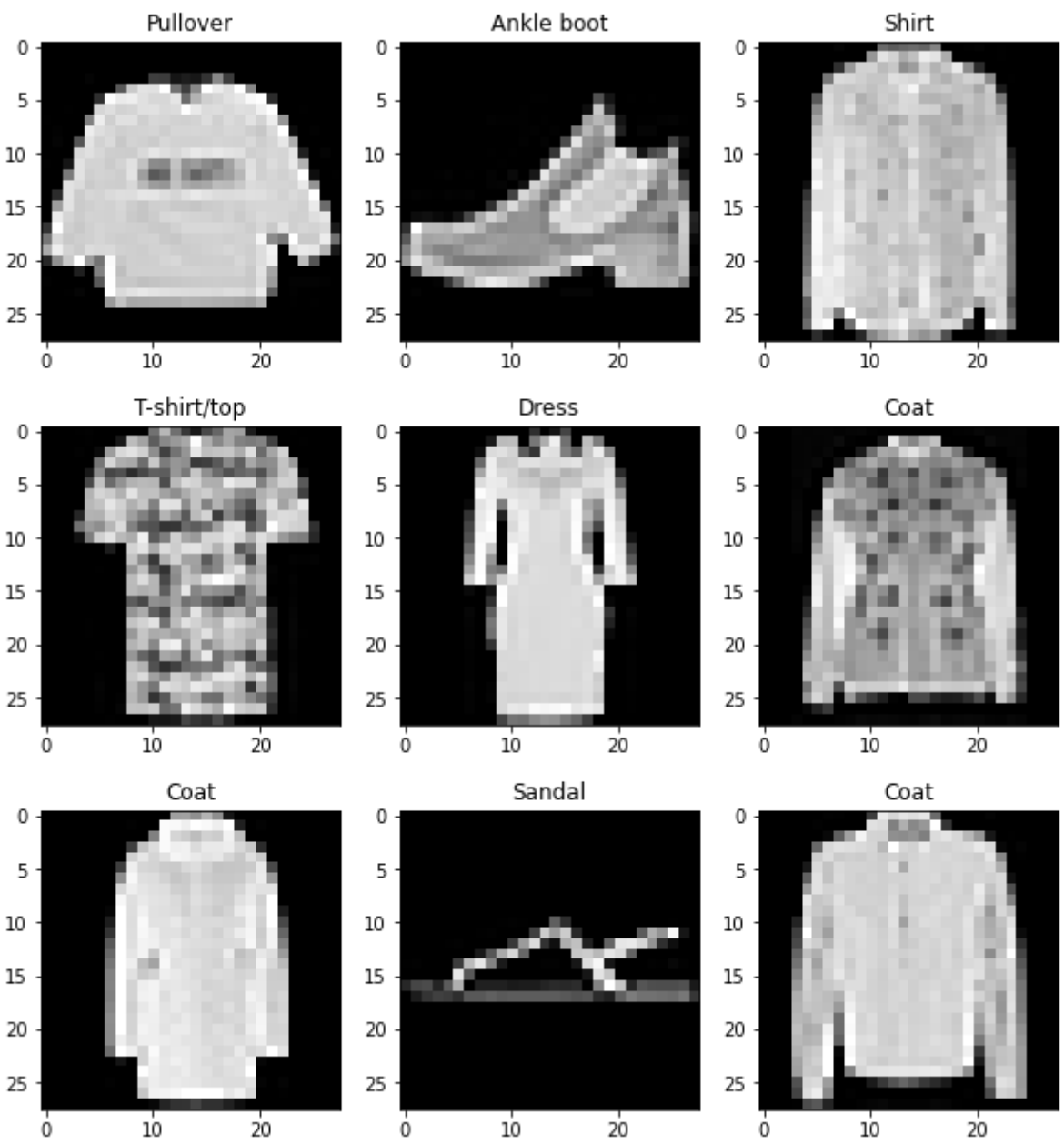
Image Pre processing

In [6]:
```
#extract the labels from the training and test set
train_labels = train_data['label'].to_numpy()
test_labels = test_data['label'].to_numpy()

#remove the labels from the training and test sets
train_vect = train_data.iloc[: , 1:].to_numpy()
test_vect = test_data.iloc[: , 1:].to_numpy()


print('Shape of traning data: '+ str(train_vect.shape))
print('Shape of test data: '+ str(test_vect.shape))
```

```
Shape of traning data: (60000, 784)
Shape of test data: (10000, 784)
```

In [7]:
```
#display 9 images from training set to get an idea
reshaped_train = train_vect.reshape(-1,28,28,1)
print(reshaped_train.shape)

class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat','Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot'
]
plt.gray() # B/W Images
plt.figure(figsize = (10,11))
for i in range(9):
 plt.subplot(3,3,i+1)
 plt.title(class_names[train_labels[i]] )
 plt.imshow(np.array(reshaped_train[i]).reshape((28,28)))
 plt.grid(False)
```

```
(60000, 28, 28, 1)

<Figure size 432x288 with 0 Axes>
```



Task 2.a) ANN: Initially train a classifier using artificial neural network while treating pixels as different features

- Input Layer: The Input Layer provides an entry point for incoming data. As such it needs to match the format or "shape" of the expected input. For this data set the input must be of 28 x 28 x 1 since it's grayscale and has 784 individual pixels for each image.
- Hidden Layer: Hidden layers are so called because they sit between the Input and Output Layers and have no contact with the "outside world". Their role is to identify features from the input data and use these to correlate between a given input and the correct output. An ANN can have multiple Hidden Layers.
- Output Layer: The Output Layer delivers the end result from the ANN and is structured according to the use case you are working on. For this dataset since we want an ANN to recognise 10 different objects in images we want 10 output nodes, each representing one of the objects we are trying to find. The final score from each output node would then indicate whether or not the associated object had been found by the ANN.

In [8]:
```
#Normalize the test and training data set to have values between 0-1

X_train = train_vect.astype(float) / 255.
Y_train = train_labels

X_test = test_vect.astype(float) / 255.
Y_test = test_labels
```

In [9]:
```
from tensorflow import keras

#ANN with one hidden layer
ANNmodel = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(784,)), #Input layer of 784 pixels
    tf.keras.layers.Dense(128, activation='relu'), # Dense layer with activation function ReLU
    tf.keras.layers.Dense(10) #Output layer with 10 nodes to classify images to 10 classes
])
# Display a summary of the model structure
ANNmodel.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 128)               100480

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
```

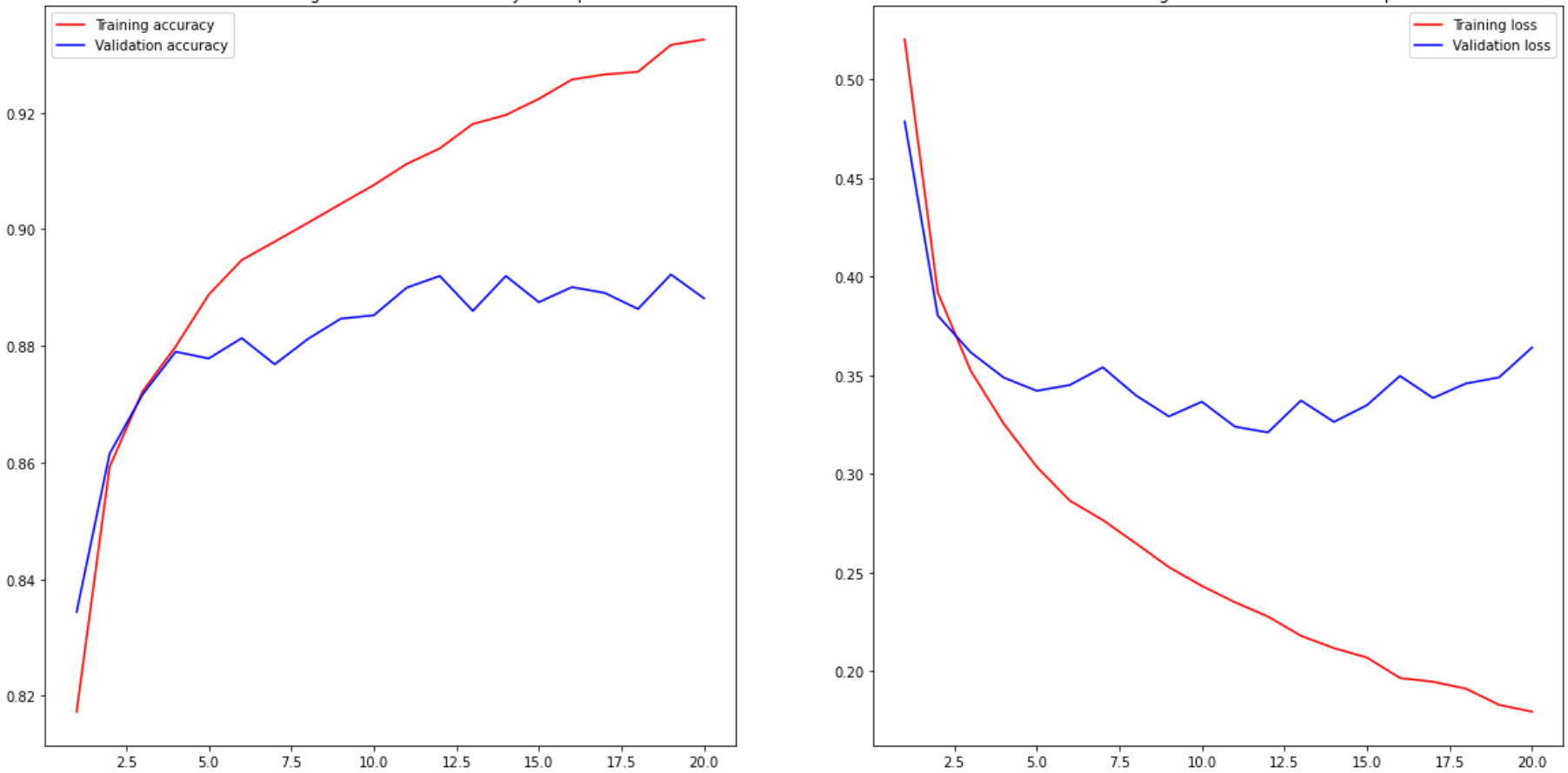
In [10]:
```
#Compile the model with
# optimizer as Adam algorithm,
# loss function :  Sparse Categorical Crossentropy
# metrics : accuracy
ANNmodel.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

In [11]:
```
#Fit the model to the training set using 20 epochs and take 0.2 of the training
#data to validate the model
ANNhistory = ANNmodel.fit(X_train, Y_train, epochs=20,validation_split=0.2)
```

```
Epoch 1/20
1500/1500 [==============================] - 14s 7ms/step - loss: 0.5202 - accuracy: 0.8172 - val_loss: 0.4786 - val_
accuracy: 0.8343
Epoch 2/20
1500/1500 [==============================] - 6s 4ms/step - loss: 0.3919 - accuracy: 0.8592 - val_loss: 0.3802 - val_a
ccuracy: 0.8615
Epoch 3/20
1500/1500 [==============================] - 5s 3ms/step - loss: 0.3522 - accuracy: 0.8722 - val_loss: 0.3615 - val_a
ccuracy: 0.8717
Epoch 4/20
1500/1500 [==============================] - 5s 3ms/step - loss: 0.3253 - accuracy: 0.8799 - val_loss: 0.3487 - val_a
ccuracy: 0.8790
Epoch 5/20
1500/1500 [==============================] - 5s 3ms/step - loss: 0.3035 - accuracy: 0.8888 - val_loss: 0.3421 - val_a
ccuracy: 0.8778
Epoch 6/20
1500/1500 [==============================] - 5s 3ms/step - loss: 0.2864 - accuracy: 0.8947 - val_loss: 0.3450 - val_a
ccuracy: 0.8813
Epoch 7/20
1500/1500 [==============================] - 6s 4ms/step - loss: 0.2765 - accuracy: 0.8979 - val_loss: 0.3539 - val_a
ccuracy: 0.8768
Epoch 8/20
1500/1500 [==============================] - 5s 4ms/step - loss: 0.2647 - accuracy: 0.9011 - val_loss: 0.3398 - val_a
ccuracy: 0.8812
Epoch 9/20
1500/1500 [==============================] - 5s 3ms/step - loss: 0.2527 - accuracy: 0.9044 - val_loss: 0.3291 - val_a
ccuracy: 0.8847
Epoch 10/20
1500/1500 [==============================] - 6s 4ms/step - loss: 0.2431 - accuracy: 0.9076 - val_loss: 0.3365 - val_a
ccuracy: 0.8852
Epoch 11/20
1500/1500 [==============================] - 5s 3ms/step - loss: 0.2349 - accuracy: 0.9112 - val_loss: 0.3239 - val_a
ccuracy: 0.8900
Epoch 12/20
1500/1500 [==============================] - 5s 3ms/step - loss: 0.2276 - accuracy: 0.9139 - val_loss: 0.3210 - val_a
ccuracy: 0.8920
Epoch 13/20
1500/1500 [==============================] - 5s 3ms/step - loss: 0.2179 - accuracy: 0.9181 - val_loss: 0.3371 - val_a
ccuracy: 0.8860
Epoch 14/20
1500/1500 [==============================] - 5s 4ms/step - loss: 0.2116 - accuracy: 0.9196 - val_loss: 0.3263 - val_a
ccuracy: 0.8920
Epoch 15/20
1500/1500 [==============================] - 5s 3ms/step - loss: 0.2068 - accuracy: 0.9224 - val_loss: 0.3348 - val_a
ccuracy: 0.8875
Epoch 16/20
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1965 - accuracy: 0.9257 - val_loss: 0.3496 - val_a
ccuracy: 0.8901
Epoch 17/20
1500/1500 [==============================] - 5s 4ms/step - loss: 0.1946 - accuracy: 0.9266 - val_loss: 0.3384 - val_a
ccuracy: 0.8891
Epoch 18/20
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1910 - accuracy: 0.9270 - val_loss: 0.3458 - val_a
ccuracy: 0.8863
Epoch 19/20
1500/1500 [==============================] - 6s 4ms/step - loss: 0.1828 - accuracy: 0.9316 - val_loss: 0.3489 - val_a
ccuracy: 0.8923
Epoch 20/20
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1794 - accuracy: 0.9326 - val_loss: 0.3640 - val_a
ccuracy: 0.8882
```

In [12]:
```
#Plot the loss and accuracy of the training process

# get the training, validation accuracy and loss
acc = ANNhistory.history['accuracy']
val_acc = ANNhistory.history['val_accuracy']
loss = ANNhistory.history['loss']
val_loss = ANNhistory.history['val_loss']
epochs = range(1, len(acc) + 1)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 10))

# plotting the training and the validation accuracies
ax1.plot(epochs, acc, 'r', label='Training accuracy')
ax1.plot(epochs, val_acc, 'b', label='Validation accuracy')
ax1.set_title('Variation of Training and validation accuracy with epochs')
ax1.legend()

# plotting the training and the validation losses
ax2.plot(epochs, loss, 'r', label='Training loss')
ax2.plot(epochs, val_loss, 'b', label='Validation loss')
ax2.set_title('Variation of Training and validation loss with epochs')
ax2.legend()
```

Out[12]: <matplotlib.legend.Legend at 0x7f368012e490>



In [13]:
```
#Evaluate the model against test dataset
test_loss, test_acc = ANNmodel.evaluate(X_test, Y_test, verbose=2)

print('\nTest accuracy:', test_acc)
```

```
313/313 - 1s - loss: 0.3401 - accuracy: 0.8887 - 1s/epoch - 4ms/step

Test accuracy: 0.888700008392334
```

It turns out that the accuracy on the test dataset is a little less than the accuracy on the training dataset. This gap between training accuracy and test accuracy represents overfitting. Overfitting happens when a machine learning model performs worse on new, previously unseen inputs than it does on the training data. An overfitted model "memorizes" the noise and details in the training dataset to a point where it negatively impacts the performance of the model on the new data.

In [25]:
```
#Convert the output values of the model to probabilities to obtain the probability
# of an input belonging to each class
probability_model = tf.keras.Sequential([ANNmodel,tf.keras.layers.Softmax()])
predictions = probability_model.predict(X_test)
np.argmax(predictions[0])
```

Out[25]: 0

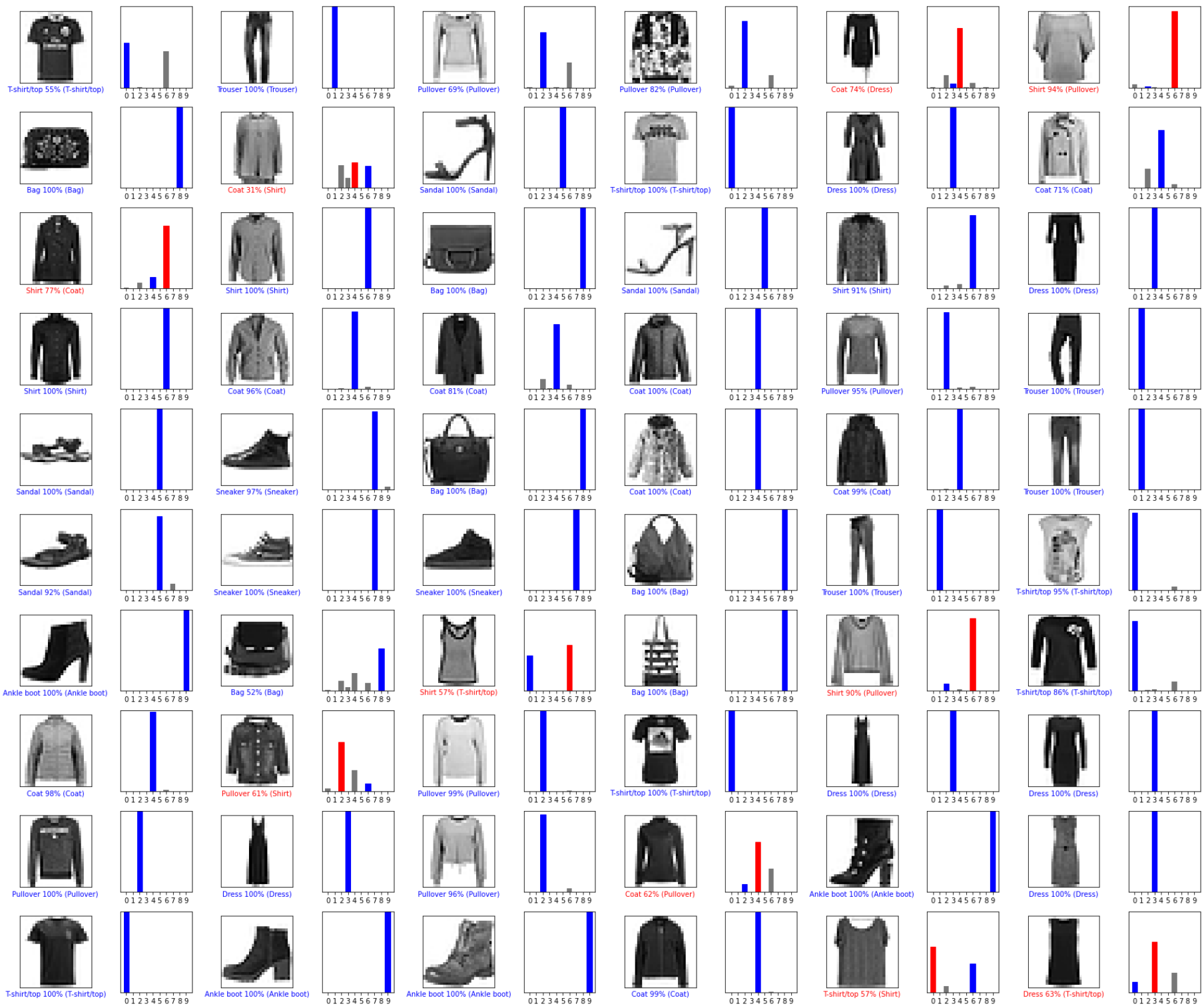
In [15]:
```
#Get the quality of the predictions obtained from the ANN model using
#sklearn classification report
from sklearn.metrics import classification_report
predicted_classes = np.argmax(predictions, axis=-1)
print(classification_report(Y_test, predicted_classes, target_names=class_names))
```

```
              precision    recall  f1-score   support

 T-shirt/top       0.85      0.84      0.84      1000
     Trouser       0.98      0.98      0.98      1000
    Pullover       0.89      0.74      0.81      1000
       Dress       0.88      0.92      0.90      1000
        Coat       0.81      0.82      0.82      1000
      Sandal       0.98      0.93      0.95      1000
       Shirt       0.68      0.76      0.72      1000
     Sneaker       0.92      0.96      0.94      1000
         Bag       0.98      0.97      0.98      1000
  Ankle boot       0.94      0.96      0.95      1000

    accuracy                           0.89     10000
   macro avg       0.89      0.89      0.89     10000
weighted avg       0.89      0.89      0.89     10000
```

- Precision: The percentage of predictions that were correct for each class. It can be seen that Trousers and Bags were predicted with 98% precision.
- Recall: The percentage of correct images identified by model. It can be seen that 98% of the images in the Trouser category were identified by the model.
- F1-score: The percentage of positive predictions that were correct.

2*(Recall* Precision) / (Recall + Precision) Trousers have the highest f1-score of 0.98

In [16]:
```
# Plot the first 60 test images, their predicted labels, and the true labels.
# correct predictions are in blue and incorrect are predictions in red.

reshape_test = test_vect.reshape(-1,28,28) #obtain the 28 x 28 images
num_rows = 10
num_cols = 6
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
  plt.subplot(num_rows, 2*num_cols, 2*i+1)
  plot_image(i,predictions[i], Y_test,reshape_test)
  plt.subplot(num_rows, 2*num_cols, 2*i+2)
  plot_value_array(i, predictions[i], Y_test)
plt.tight_layout()
plt.show()
```



Task 2.b) CNN: Train a Convolutional neural network(CNN) for the above data set considering data points as images.

- Maintain Spatial Integrity of Input Images: Images are fed into a CNN as grid of pixel values - this ensures that features spanning multiple pixels are maintained. Input for this data set is a 2D grid of 28 x 28 x 1.
- Feature Extraction Through Convolutional Filters: Convolutional filters produce feature maps that accentuate or dampen specific features in our input images.
- Feature Map Enhancement via ReLU: Reduces "noise" in feature maps and further enhances relevant features by replacing all negative activation values in each feature map with 0.
- Dimensionality Reduction via Pooling: Reduces processing requirements by reducing the size of feature maps - hopefully without loss of information.

Pre processing

In [17]:
```
#Convert the training and test vactors into images of size 28 x 28
CNN_Xtrain = train_vect.reshape(-1,28,28)
CNN_Ytrain = train_labels

CNN_Xtest = test_vect.reshape(-1,28,28)
CNN_Ytest = test_labels



print(CNN_Xtrain.shape)
print(CNN_Ytrain.shape)
print(CNN_Xtest.shape)
print(CNN_Ytest.shape)
```

```
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
```

The Convolutional Neural Network model

The CNN model is built to achieve two tasks.

1. Feature extraction:
    - Extract specific features of the images by applying 32 convolutional filters of size 3x3 (Conv2D).
    - Then, the dimension reduction of feature maps without loss of information is done using MaxPooling2D to speed up computations.
    - Afterwards, a Dropout Layer is added to reduce overfitting and force the model to learn mulitiple representations of the same data by randomly disabling a given amount of neurons in the learning phase.
    - To extract the features of the processed feature maps, another covolutional layer with 32 filters is used alongwith a MaxPooling layer and a dropout layer
2. Classification: The identified features are mapped to a class out of the 10 classes. This is done in the two Dense Layers.

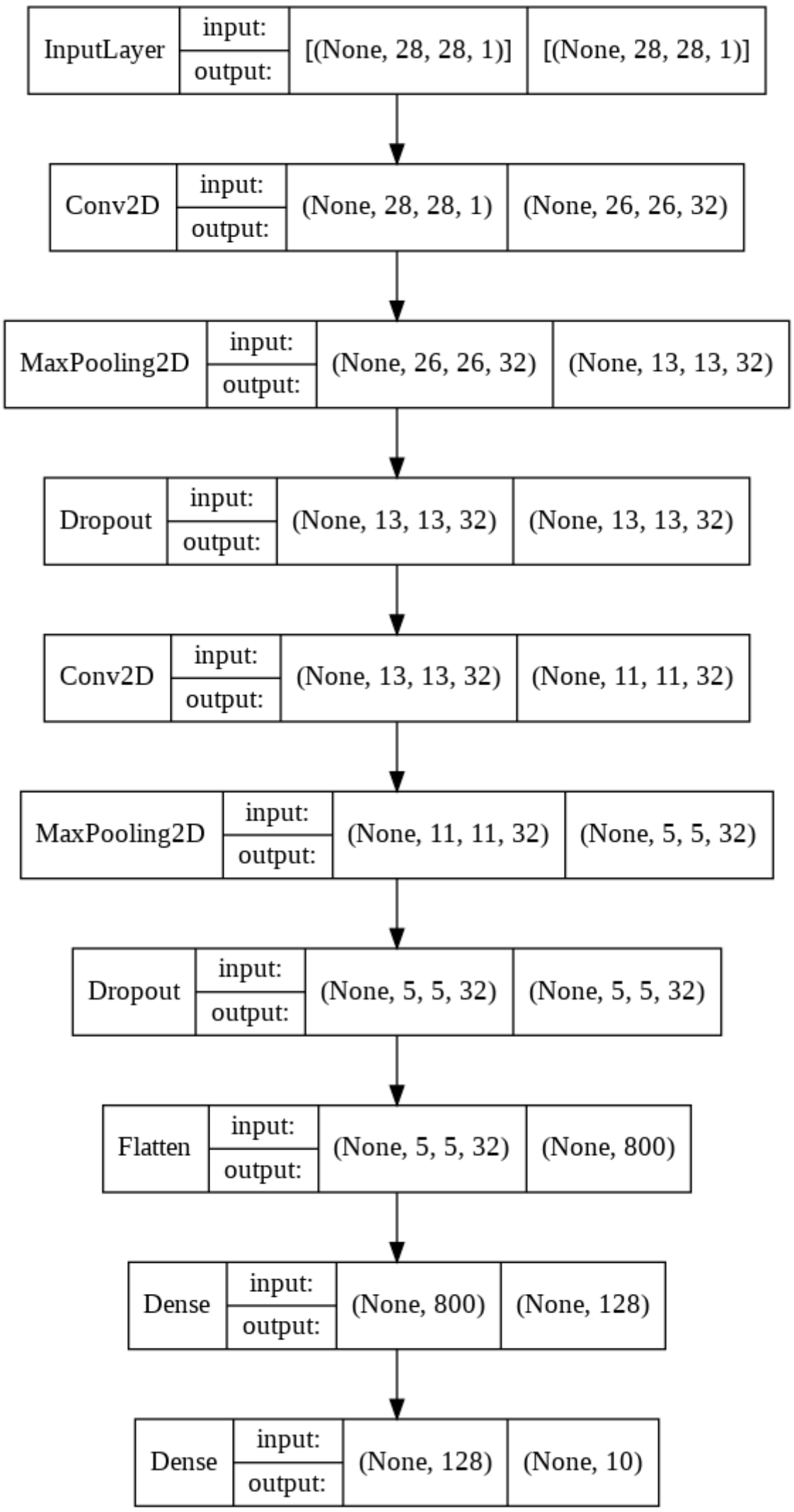
In [18]:
```
# Begin definine the model by defining an empty stack
CNNmodel = tf.keras.models.Sequential()

#LAYER 1
# Add 32 convolutional filters as the first layer.
# results in 32 feature maps of size 26 x 26 because filters are applied without padding.
CNNmodel.add(
    tf.keras.layers.Conv2D(
        filters=32, # How many filters model will learn
        kernel_size=(3, 3), # Size of feature map that will slide over image
        strides=(1, 1), # How the feature map "steps" across the image
        padding='valid', # the model is not using padding
        activation='relu', # Rectified Linear Unit Activation Function
        input_shape=(28, 28, 1) # The expected input shape for this layer
    )
)

#LAYER 2
# reduce the dimensionality of each feature to which reduces the number of
# parameters that the model needs to learn using MaxPooling2D.
# Select the maximum value from each 2x2 group of pixels.
# Stride is set to 2 to ensure that each pixel is only sampled once.
# Reduce size to 13 x 13 x 32

CNNmodel.add(
    tf.keras.layers.MaxPooling2D(
        pool_size=(2, 2), # Size feature will be mapped to
        strides=(2, 2) # How the pool "steps" across the feature
    )
)

#LAYER 3
# Resist overfitting using a dropout layer.
# randomly disable 25% of it's nodes at a given time
CNNmodel.add(
    tf.keras.layers.Dropout(
        rate=0.25 # Randomly disable 25% of neurons
    )
)


#LAYER 4
# Add 32 convolutional filters to filter features of the feature maps.
# results in 32 feature maps of size 26 x 26 because filters are applied without padding.
CNNmodel.add(
    tf.keras.layers.Conv2D(
        filters=32, # How many filters the model will learn
        kernel_size=(3, 3), # Size of feature map that will slide over image
        strides=(1, 1), # How the feature map "steps" across the image
        padding='valid', # not using padding
        activation='relu', # Rectified Linear Unit Activation Function
        input_shape=(28, 28, 1) # The expected input shape for this layer
    )
)

#LAYER 5
# reduce the dimensionality of each feature to which reduces the number of
# parameters that the model needs to learn using MaxPooling2D.
# Select the maximum value from each 2x2 group of pixels.
# Stride is set to 2 to ensure that each pixel is only sampled once.
# Reduce size to 5 x 5 x 32

CNNmodel.add(
    tf.keras.layers.MaxPooling2D(
        pool_size=(2, 2), # Size feature will be mapped to
        strides=(2, 2) # How the pool "steps" across the feature
    )
)


#LAYER 6
# Resist overfitting using a dropout layer.
# randomly disable 25% of it's nodes at a given time
CNNmodel.add(
    tf.keras.layers.Dropout(
        rate=0.25 # Randomly disable 25% of neurons
    )
)

#LAYER 7
# Flatten the output of dropout layer to a 1D list of 800 nodes (5 x 5 x 32)
# that can be sent to the Dense layers for classification
CNNmodel.add(
    tf.keras.layers.Flatten()
)

#LAYER 8
# First dense layer used to map the 800 nodes to 128 nodes, using a ReLU activation function
```

```
CNNmodel.add(
    tf.keras.layers.Dense(
        units=128, # Output shape
        activation='relu' # Rectified Linear Unit Activation Function
    )
)

#LAYER 9: Final layer
# Final layer with 10 outputs and a softmax activation.
# Softmax activation is used to calculate the output based on the probabilities.
# Each class is assigned a probability and the class with the maximum
# probability is the model's output for the input.
CNNmodel.add(
    tf.keras.layers.Dense(
        units=10, # Output shape
        activation='softmax' # Softmax Activation Function
    )
)

# Build the model
CNNmodel.compile(
    loss=tf.keras.losses.sparse_categorical_crossentropy, # loss function
    optimizer=tf.keras.optimizers.Adam(), # optimizer function
    metrics=['accuracy'] # reporting metric
)

# Display a summary of the model structure
CNNmodel.summary()

print('\nLayers of the Model')
#visualize the model structure
tf.keras.utils.plot_model(CNNmodel,show_shapes=True, show_layer_names=False)
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 dropout (Dropout)           (None, 13, 13, 32)        0

 conv2d_1 (Conv2D)           (None, 11, 11, 32)        9248

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 32)         0
 2D)

 dropout_1 (Dropout)         (None, 5, 5, 32)          0

 flatten_1 (Flatten)         (None, 800)               0

 dense_2 (Dense)             (None, 128)               102528

 dense_3 (Dense)             (None, 10)                1290

=================================================================
Total params: 113,386
Trainable params: 113,386
Non-trainable params: 0
_________________________________________________________________
```

Layers of the Model

Out[18]:



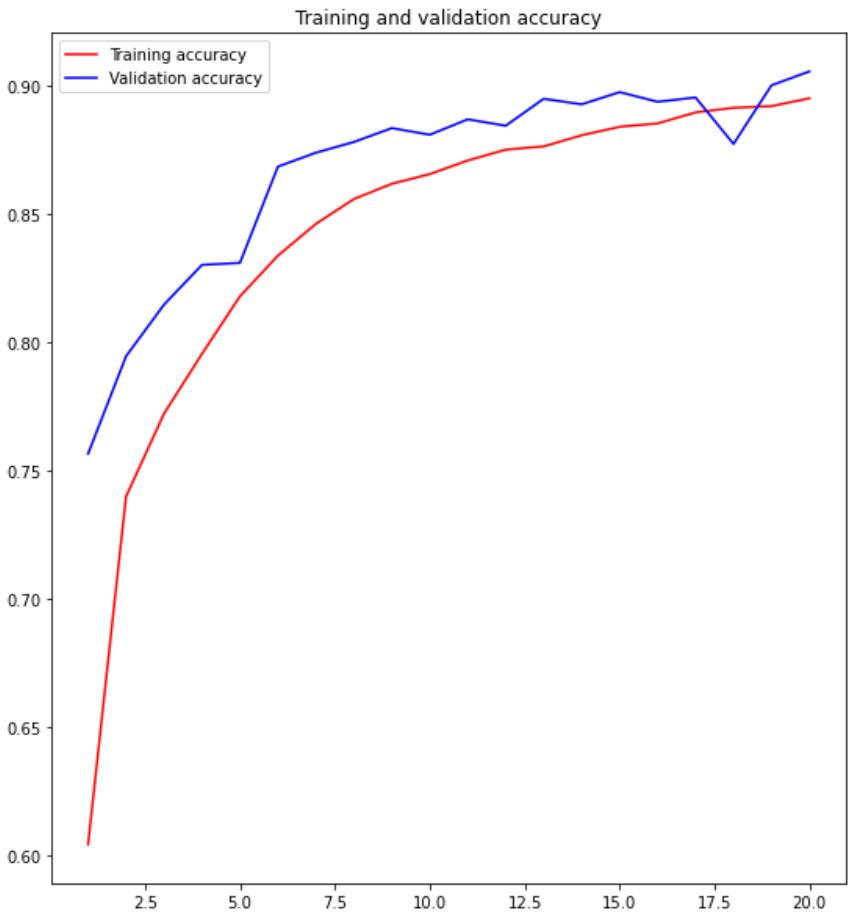
Train the model

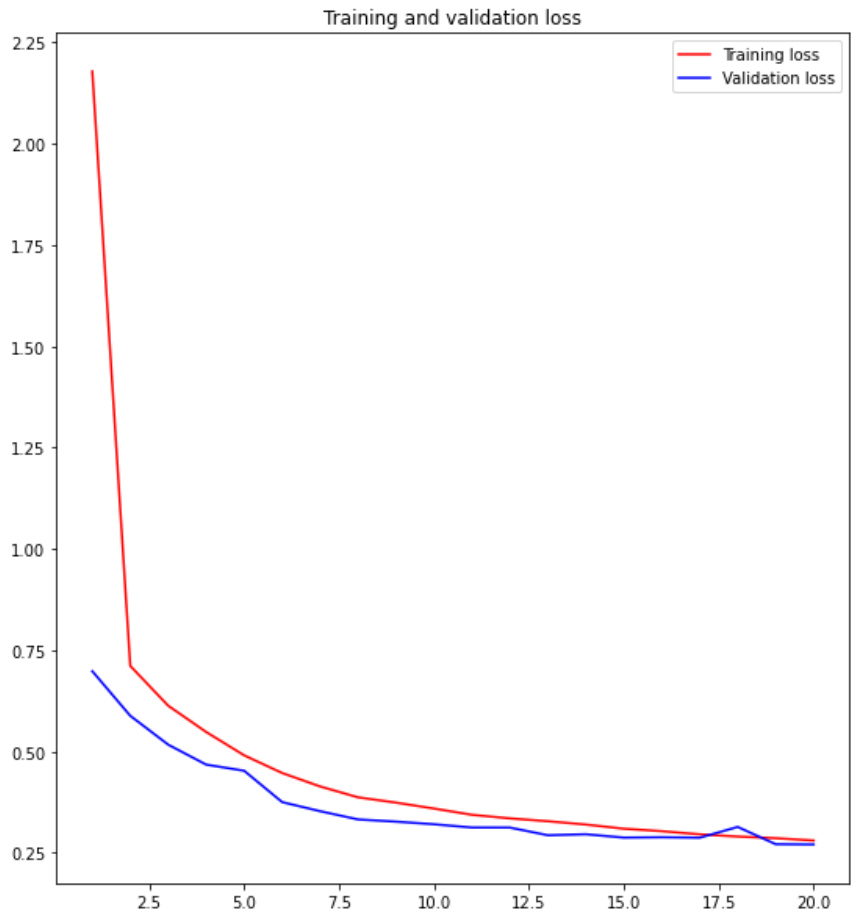
In [19]:
```
# Add an empty color dimension as the Convolutional net is expecting this
x_train = np.expand_dims(CNN_Xtrain, -1)
x_test = np.expand_dims(CNN_Xtest, -1)

# Train the CNN on the training data
CNNhistory = CNNmodel.fit(

      # Training data : features (images) and classes.
      x_train, CNN_Ytrain,

      # number of samples to work through before updating the
      # internal model parameters via back propagation.
      batch_size=256,

      # An epoch is an iteration over the entire training data.
      epochs=20,

      # The model will set apart his fraction of the training
      # data, will not train on it, and will evaluate the loss
      # and any model metrics on this data at the end of
      # each epoch.
      validation_split=0.2,

      verbose=1)
```

```
Epoch 1/20
188/188 [==============================] - 10s 15ms/step - loss: 2.1789 - accuracy: 0.6045 - val_loss: 0.6984 - val_a
ccuracy: 0.7566
Epoch 2/20
188/188 [==============================] - 2s 12ms/step - loss: 0.7114 - accuracy: 0.7400 - val_loss: 0.5883 - val_ac
curacy: 0.7946
Epoch 3/20
188/188 [==============================] - 2s 13ms/step - loss: 0.6132 - accuracy: 0.7722 - val_loss: 0.5170 - val_ac
curacy: 0.8147
Epoch 4/20
188/188 [==============================] - 2s 13ms/step - loss: 0.5483 - accuracy: 0.7955 - val_loss: 0.4678 - val_ac
curacy: 0.8302
Epoch 5/20
188/188 [==============================] - 2s 13ms/step - loss: 0.4907 - accuracy: 0.8179 - val_loss: 0.4527 - val_ac
curacy: 0.8309
Epoch 6/20
188/188 [==============================] - 2s 12ms/step - loss: 0.4471 - accuracy: 0.8337 - val_loss: 0.3752 - val_ac
curacy: 0.8684
Epoch 7/20
188/188 [==============================] - 2s 12ms/step - loss: 0.4139 - accuracy: 0.8461 - val_loss: 0.3528 - val_ac
curacy: 0.8738
Epoch 8/20
188/188 [==============================] - 2s 13ms/step - loss: 0.3870 - accuracy: 0.8558 - val_loss: 0.3325 - val_ac
curacy: 0.8780
Epoch 9/20
188/188 [==============================] - 2s 12ms/step - loss: 0.3740 - accuracy: 0.8617 - val_loss: 0.3270 - val_ac
curacy: 0.8834
Epoch 10/20
188/188 [==============================] - 2s 12ms/step - loss: 0.3593 - accuracy: 0.8655 - val_loss: 0.3206 - val_ac
curacy: 0.8808
Epoch 11/20
188/188 [==============================] - 2s 13ms/step - loss: 0.3439 - accuracy: 0.8708 - val_loss: 0.3125 - val_ac
curacy: 0.8868
Epoch 12/20
188/188 [==============================] - 2s 12ms/step - loss: 0.3351 - accuracy: 0.8750 - val_loss: 0.3124 - val_ac
curacy: 0.8843
Epoch 13/20
188/188 [==============================] - 2s 12ms/step - loss: 0.3280 - accuracy: 0.8763 - val_loss: 0.2934 - val_ac
curacy: 0.8948
Epoch 14/20
188/188 [==============================] - 2s 13ms/step - loss: 0.3199 - accuracy: 0.8806 - val_loss: 0.2959 - val_ac
curacy: 0.8927
Epoch 15/20
188/188 [==============================] - 2s 12ms/step - loss: 0.3096 - accuracy: 0.8839 - val_loss: 0.2876 - val_ac
curacy: 0.8974
Epoch 16/20
188/188 [==============================] - 2s 13ms/step - loss: 0.3035 - accuracy: 0.8852 - val_loss: 0.2885 - val_ac
curacy: 0.8937
Epoch 17/20
188/188 [==============================] - 2s 13ms/step - loss: 0.2957 - accuracy: 0.8895 - val_loss: 0.2874 - val_ac
curacy: 0.8953
Epoch 18/20
188/188 [==============================] - 2s 12ms/step - loss: 0.2903 - accuracy: 0.8913 - val_loss: 0.3140 - val_ac
curacy: 0.8773
Epoch 19/20
188/188 [==============================] - 3s 15ms/step - loss: 0.2861 - accuracy: 0.8920 - val_loss: 0.2713 - val_ac
curacy: 0.9001
Epoch 20/20
188/188 [==============================] - 2s 12ms/step - loss: 0.2805 - accuracy: 0.8950 - val_loss: 0.2707 - val_ac
curacy: 0.9055
```

In [20]:
```
# extracting the return values after training the model
acc = CNNhistory.history['accuracy']
val_acc = CNNhistory.history['val_accuracy']
loss = CNNhistory.history['loss']
val_loss = CNNhistory.history['val_loss']

epochs = range(1, len(acc) + 1)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 10))
# plotting the accuracy of the model against training and validation data
ax1.plot(epochs, acc, 'r', label='Training accuracy')
ax1.plot(epochs, val_acc, 'b', label='Validation accuracy')
ax1.set_title('Training and validation accuracy')
ax1.legend()
# plotting the loss of the model against training and validation data
ax2.plot(epochs, loss, 'r', label='Training loss')
ax2.plot(epochs, val_loss, 'b', label='Validation loss')
ax2.set_title('Training and validation loss')
ax2.legend()
```

Out[20]: <matplotlib.legend.Legend at 0x7f3609268110>





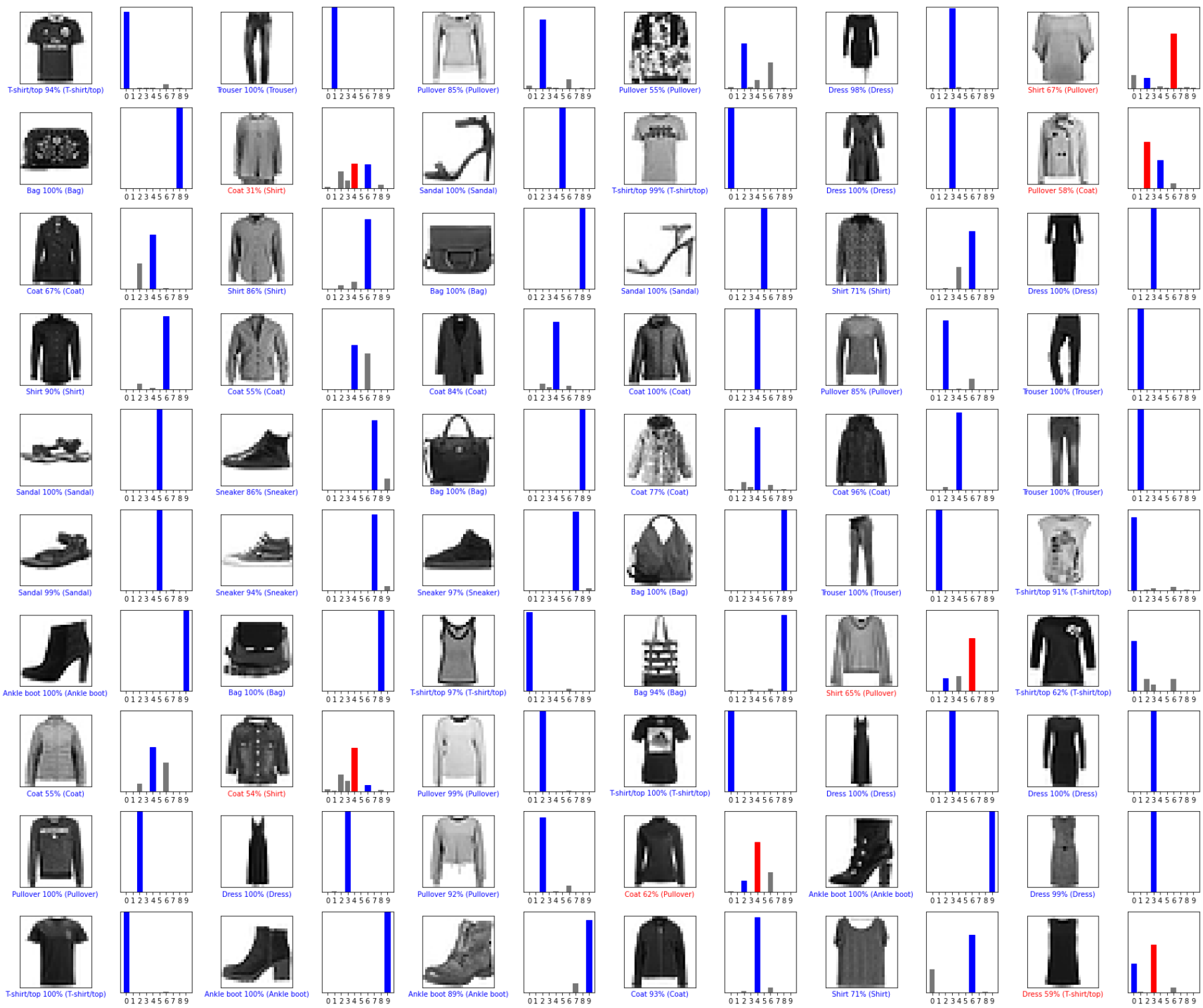
In [21]:
```
#Get the quality of the predictions obtained from the CNN model using
#sklearn classification report
from sklearn.metrics import classification_report
predicted_classes = np.argmax(CNNmodel.predict(x_test), axis=-1)
print(classification_report(CNN_Ytest, predicted_classes, target_names=class_names))
```

```
              precision    recall  f1-score   support

 T-shirt/top       0.84      0.89      0.87      1000
     Trouser       0.99      0.98      0.99      1000
    Pullover       0.86      0.82      0.84      1000
       Dress       0.89      0.94      0.92      1000
        Coat       0.85      0.85      0.85      1000
      Sandal       0.99      0.96      0.98      1000
       Shirt       0.76      0.70      0.73      1000
     Sneaker       0.95      0.94      0.95      1000
         Bag       0.96      0.99      0.97      1000
  Ankle boot       0.94      0.98      0.96      1000

    accuracy                           0.91     10000
   macro avg       0.91      0.91      0.91     10000
weighted avg       0.91      0.91      0.91     10000
```

- Precision: The percentage of predictions that were correct for each class. It can be seen that sandals and Bags were predicted with 97% precision, and trousers were identified with a precision of 99%
- Recall: The percentage of correct images identified by model. It can be seen that 99% of the images in the Bag category were identified by the model.
- F1-score: The percentage of positive predictions that were correct.

  2*(Recall* Precision) / (Recall + Precision) Trousers have the highest f1-score of 0.99

In [22]:
```
CNNpredictions = CNNmodel.predict(x_test)

# Plot the first 60 test images, their predicted labels, and the true labels.
# Color correct predictions in blue and incorrect predictions in red.
num_rows = 10
num_cols = 6
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
  plt.subplot(num_rows, 2*num_cols, 2*i+1)
  plot_image(i,CNNpredictions[i], CNN_Ytest,reshape_test)
  plt.subplot(num_rows, 2*num_cols, 2*i+2)
  plot_value_array(i, CNNpredictions[i], CNN_Ytest)
plt.tight_layout()
plt.show()
```



c. Identify the difference between above 2 models

1. CNN model maintains the spatial integrity of the images where as ANN model does not.

- The main difference is that the ANN model used in the a) part has an input layer of a 1D list of 784 pixels in each image, where as the CNN model takes the a 2D grid of size 28 x 28 (image as a whole) in the input layer.
- As a result, the CNN model maintains the Spatial integrity of the input images.

1. CNN model extract the features of the image through convolutional filters

- In the CNN model, I have used 32 2D convolutional filters of size 3x3 to extract different features (such as edges) of an input image, to learn the features specific to each image category. These weights (filters) are refined by back-propogation during the training phase.
- This component is not used in the ANN model used in part a) as that model takes a 1D array of pixels and cannot extract features.

1. Feature maps are enhanced by the ReLU activation function in CNN model.

- ReLU function is used to ensure that only nodes with positive weights will send their values to the proceeding layers. This leads to less processing in the proceeding layers, and less overfitting.

1. CNN model reduces dimensions of the images using pooling.

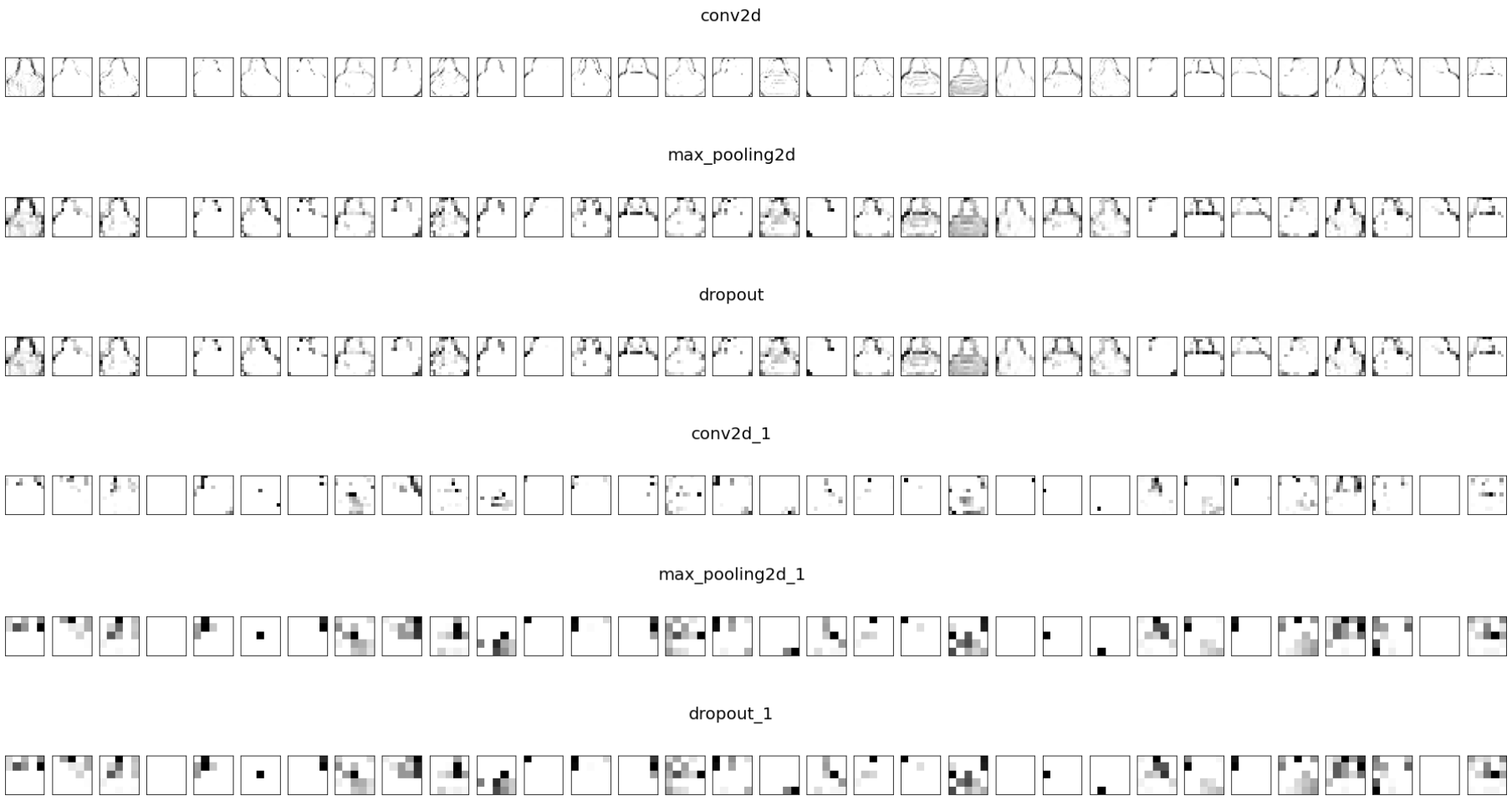
- The feature map size is reduced without information loss to reduce the amount of processing required and save time and resources used for training.
- Since ANN doesn't deal with feature maps or 2D inputs, it require such optimizations.
- Both ANN and CNN models approach the output layer by gradually reducing the number of nodes in the Dense Layers.

1. CNN model converges to better accuracy faster than ANN model, with less overfitting.

- As observed in the training and validation accuracies,it can be seen that the ANN model leads to an overfitting model. Even though drop out layers can be used to reduce the overfitting, it leads to a reduction on prediction accuracy in the ANN model. It was observed that the higher number of layers and higher number of epochs again leads to overfitting.
- However, the CNN model could be trained to classify the images with much less overfitting and it managed to obtain an accuracy closer to 90% withing 20 epochs.

d. Visualize the different layers in the CNN and identify different patterns recognized in each layer of the network.

In [24]:
```
# Store the names of the layers that must observed seperately
layer_names =[]

for layer in CNNmodel.layers[:6]:
    layer_names.append(layer.name) # Names of the layers

# Get the list of layers for the existing model
layer_dict = {layer.name : layer for layer in CNNmodel.layers}

# Choose an image from the training set
image = CNN_Xtrain[20]

# Add the extra dimension expected by the slice
image = np.expand_dims(image, axis=0)
for layer_num in range (len(layer_names)):
  # Create a copy of the existing model containing just the specific layer Layer
  modelslice = tf.keras.Model(inputs=CNNmodel.inputs, outputs=layer_dict[layer_names[layer_num]].output)

  # Send the image through the model with the specific layer
  feature_maps = modelslice.predict(image)

  fig = plt.figure(figsize=(32, 3))
  fig.suptitle('\n'+layer_names[layer_num], fontsize=20)

  # Display the 32 feature maps from each layer
  for i in range(32):
      plt.subplot(1,32,i+1)
      plt.xticks([])
      plt.yticks([])
      plt.imshow(feature_maps[0, :, :, i-1],plt.cm.binary)

plt.show()
```



- Conv2d: The Conv2D Layer shows the images undergoing 32 filters, each one sized as 3x3 pixels with a stride of 1. Since padding is not used the images are 26 x 26. Each feature map shows slightly different features of an input image (different edges and characteristics) further enhance by the ReLU Activation Function.
- max_pooling2d: The MaxPooling Layer downsamples the feature maps,to reduce processing requirements while also scaling down the total number of parameters that the model needs to learn. The layer downsamples the feature maps to 13 x 13 images.
- dropout: The Dropout Layer fights overfitting and forces the model to learn mulitiple representations of the same data by randomly disabling a given amount of neurons in the learning phase.

e. Identify the change of complexity of the patterns recognized by layers with the depth of the network.

Multiple layers of convolutional filter can be used in the model to identify different features of the outputs coming from a specific layer.(Refer to the plot in part d that shows the different features extracted by each layer of conv2d)

When a convolutional filter is applied to the raw image, it will extract the basic/low-level features such as edges of the image. This convolutional filters can then again be applied after reducing dimensions (maxpooling), dropout layers or even another convolutional layer to identify features hidden in more depth. This is clearly visible in the plot in part d, because conv2d_1 has feature maps highlighting features that are not recognized by conv2d layer.

Therefore, it can be said that the complexity of the patterns recognized by layers increase as the depth increase.

f. Discuss having more or less nodes in a single layer and having a deep or a shallow network against the computational complexity.

The number of nodes and layers controls the learning capacity of the model. For the model to correctly train on the training set, the number of nodes and layers must be adaquet.

Effect of the number of nodes in a single layer

Less number of nodes/neurons per layer may lead to underfitting and high statistical bias while too many neurons per layer may lead to overfitting and high variance and high computational complexity. As a result, if too many neurons are used in a layer, it will take more time to train the network and on the other hand, overfitting may lead to a less accurate model.

Dropout layers are added to optimize the number of neurons in a hidden layer and increase computational performance by removing the neurons that have no impact on the performance (weights closer to zero) of the network.

Effect of the number of layers (deep or shallow network)

Increasing the depth increases the capacity of the model. Training deep models, with many hidden layers, can be computationally more efficient than training a single layer network with a vast number of nodes. By adding more layers and more units within a layer, a deep network can represent functions of increasing complexity.

Adding more layers lead to higher computational complexity and having too many layers may lead to overfitting or get stuck with a sub-optimal set of weights. Training models with more than few layers is also problemetics with issues like vanishing gradient.

g. Discuss about the way you defined the optimum neural network architecture for the above problem.

As demonstrated by part a and part b, it's clear that to recognize patterns in images and classify images a Convolutional Neural Network is the best approach out of the two because it manages to protect the spatial integrity of the images and identify classes by the placement of features and patterns in a given image without manual interference.

As discussed from part e and f, it's better to have an optimal number of nodes and hidden layers that won't lead to overfitting or sub-optimal weights that will lead to a poorly trained model. The best number of layers and nodes were identified by trial and error and finding the model that gives higher accuracy and precision for the validation and test data sets.

Therefore, the optimum model to recognize patterns of images and classify them is a Convolutional Neural Network, that is neight too deep nor shallow and has a number of parameters that are easily and quickly trainable.