From internet :

Basic classification: Classify images of clothing

This guide trains a neural network model to classify images of clothing, like sneakers and shirts. It's okay if you don't understand all the details; this is a fast-paced overview of a complete TensorFlow program with the details explained as you go.

This guide uses tf.keras, a high-level API to build and train models in TensorFlow.

In [12]:
```
# TensorFlow and tf.keras
import tensorflow as tf

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt

print(tf.__version__)
```

```
2.1.0
```

Import the Fashion MNIST dataset

This guide uses the Fashion MNIST dataset which contains 70,000 grayscale images in 10 categories. The images show individual articles of clothing at low resolution (28 by 28 pixels), as seen here:

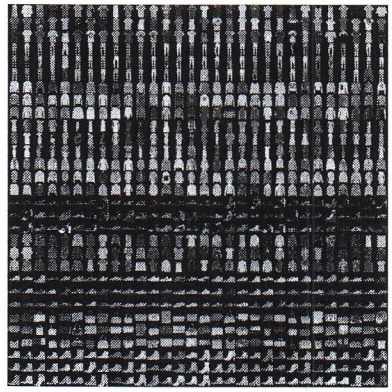


Figure 1. Fashion-MNIST samples (by Zalando, MIT License).

Fashion MNIST is intended as a drop-in replacement for the classic MNIST dataset—often used as the "Hello, World" of machine learning programs for computer vision. The MNIST dataset contains images of handwritten digits (0, 1, 2, etc.) in a format identical to that of the articles of clothing you'll use here.

This guide uses Fashion MNIST for variety, and because it's a slightly more challenging problem than regular MNIST. Both datasets are relatively small and are used to verify that an algorithm works as expected. They're good starting points to test and debug code.

Here, 60,000 images are used to train the network and 10,000 images to evaluate how accurately the network learned to classify images. You can access the Fashion MNIST directly from TensorFlow. Import and load the Fashion MNIST data directly from TensorFlow:

In [13]:
```
fashion_mnist = tf.keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

Loading the dataset returns four NumPy arrays:

- The `train_images` and `train_labels` arrays are the *training set*—the data the model uses to learn.
- The model is tested against the *test set*, the `test_images` , and `test_labels` arrays.

The images are 28x28 NumPy arrays, with pixel values ranging from 0 to 255. The *labels* are an array of integers, ranging from 0 to 9. These correspond to the *class* of clothing the image represents:

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

Each image is mapped to a single label. Since the *class names* are not included with the dataset, store them here to use later when plotting the images:

In [14]:
```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

Explore the data

Let's explore the format of the dataset before training the model. The following shows there are 60,000 images in the training set, with each image represented as 28 x 28 pixels:

In [15]:
```
train_images.shape
```

Out[15]:
```
(60000, 28, 28)
```

Likewise, there are 60,000 labels in the training set:

In [16]:
```
len(train_labels)
```

Out[16]:
```
60000
```

Each label is an integer between 0 and 9:

In [17]:
```
train_labels
```

Out[17]:
```
array([9, 0, 0, ..., 3, 0, 5], dtype=uint8)
```

There are 10,000 images in the test set. Again, each image is represented as 28 x 28 pixels:

In [18]:
```
test_images.shape
```

Out[18]:
```
(10000, 28, 28)
```

And the test set contains 10,000 images labels:

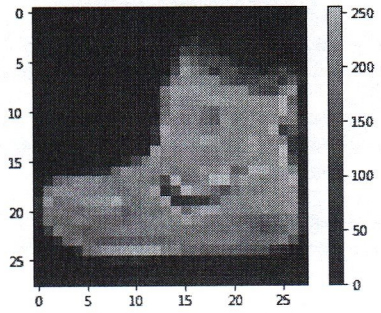
In [19]:
```
len(test_labels)
```

Out[19]:
```
10000
```

Preprocess the data

The data must be preprocessed before training the network. If you inspect the first image in the training set, you will see that the pixel values fall in the range of 0 to 255:

In [20]:
```
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
```



Scale these values to a range of 0 to 1 before feeding them to the neural network model. To do so, divide the values by 255. It's important that the *training set* and the *testing set* be preprocessed in the same way:

In [21]:
```
train_images = train_images / 255.0

test_images = test_images / 255.0
```

To verify that the data is in the correct format and that you're ready to build and train the network, let's display the first 25 images from the *training set* and display the class name below each image.

In [22]:
```
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```

Build the model

Building the neural network requires configuring the layers of the model, then compiling the model.

Set up the layers

The basic building block of a neural network is the *layer*. Layers extract representations from the data fed into them. Hopefully, these representations are meaningful for the problem at hand.

Most of deep learning consists of chaining together simple layers. Most layers, such as `tf.keras.layers.Dense` , have parameters that are learned during training.

In [23]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])
```

The first layer in this network, `tf.keras.layers.Flatten` , transforms the format of the images from a two-dimensional array (of 28 by 28 pixels) to a one-dimensional array (of 28 * 28 = 784 pixels). Think of this layer as unstacking rows of pixels in the image and lining them up. This layer has no parameters to learn; it only reformats the data.

After the pixels are flattened, the network consists of a sequence of two `tf.keras.layers.Dense` layers. These are densely connected, or fully connected, neural layers. The first `Dense` layer has 128 nodes (or neurons). The second (and last) layer returns a logits array with length of 10. Each node contains a score that indicates the current image belongs to one of the 10 classes.

Compile the model

Before the model is ready for training, it needs a few more settings. These are added during the model's *compile* step:

- *Loss function* —This measures how accurate the model is during training. You want to minimize this function to "steer" the model in the right direction.
- *Optimizer* —This is how the model is updated based on the data it sees and its loss function.
- *Metrics* —Used to monitor the training and testing steps. The following example uses *accuracy*, the fraction of the images that are correctly classified.

In [24]:
```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Train the model

Training the neural network model requires the following steps:

1. Feed the training data to the model. In this example, the training data is in the `train_images` and `train_labels` arrays.
2. The model learns to associate images and labels.
3. You ask the model to make predictions about a test set—in this example, the `test_images` array.
4. Verify that the predictions match the labels from the `test_labels` array.

Feed the model

To start training, call the `model.fit` method—so called because it "fits" the model to the training data:

In [25]:
```
model.fit(train_images, train_labels, epochs=10)
```

```
Train on 60000 samples
Epoch 1/10
60000/60000 [==============================] - 6s 96us/sample - loss: 0.4996 - accuracy: 0.8241
Epoch 2/10
60000/60000 [==============================] - 5s 77us/sample - loss: 0.3757 - accuracy: 0.8650
Epoch 3/10
60000/60000 [==============================] - 5s 77us/sample - loss: 0.3352 - accuracy: 0.8787
Epoch 4/10
60000/60000 [==============================] - 5s 84us/sample - loss: 0.3131 - accuracy: 0.8841
Epoch 5/10
60000/60000 [==============================] - 5s 77us/sample - loss: 0.2961 - accuracy: 0.8904
```

```
Epoch 6/10
60000/60000 [==============================] - 5s 80us/sample - loss: 0.2810 - accuracy: 0.8959
Epoch 7/10
60000/60000 [==============================] - 5s 84us/sample - loss: 0.2677 - accuracy: 0.9003
Epoch 8/10
60000/60000 [==============================] - 5s 76us/sample - loss: 0.2582 - accuracy: 0.9044
Epoch 9/10
60000/60000 [==============================] - 4s 72us/sample - loss: 0.2475 - accuracy: 0.9069
Epoch 10/10
60000/60000 [==============================] - 4s 73us/sample - loss: 0.2391 - accuracy: 0.9104
```

Out[25]:
```
<tensorflow.python.keras.callbacks.History at 0x1e6e3043148>
```

As the model trains, the loss and accuracy metrics are displayed. This model reaches an accuracy of about 0.91 (or 91%) on the training data.

Evaluate accuracy

Next, compare how the model performs on the test dataset:

In [26]:
```
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)

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

```
10000/10000 - 1s - loss: 0.3305 - accuracy: 0.8817

Test accuracy: 0.8817
```

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. For more information, see the following:

- Demonstrate overfitting
- Strategies to prevent overfitting

Make predictions

With the model trained, you can use it to make predictions about some images. The model's linear outputs, logits. Attach a softmax layer to convert the logits to probabilities, which are easier to interpret.

In [27]:
```
probability_model = tf.keras.Sequential([model,
                                         tf.keras.layers.Softmax()])
```

In [28]:
```
predictions = probability_model.predict(test_images)
```

Here, the model has predicted the label for each image in the testing set. Let's take a look at the first prediction:

In [29]:
```
predictions[0]
```

Out[29]:
```
array([3.2722788e-08, 5.9750688e-10, 1.3003919e-08, 1.6708194e-08,
       1.5295351e-08, 3.3631828e-03, 1.8069553e-08, 4.0134717e-02,
       1.7317571e-07, 9.5650184e-01], dtype=float32)
```

A prediction is an array of 10 numbers. They represent the model's "confidence" that the image corresponds to each of the 10 different articles of clothing. You can see which label has the highest confidence value:

In [30]:
```
np.argmax(predictions[0])
```

Out[30]:
```
9
```

So, the model is most confident that this image is an ankle boot, or `class_names[9]`. Examining the test label shows that this classification is correct:

In [31]:
```
test_labels[0]
```

Out[31]:
```
9
```

Graph this to look at the full set of 10 class predictions.

In [32]:
```
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)
```

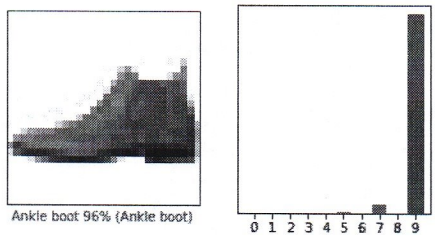
```
  thisplot[predicted_label].set_color('red')
  thisplot[true_label].set_color('blue')
```

Verify predictions

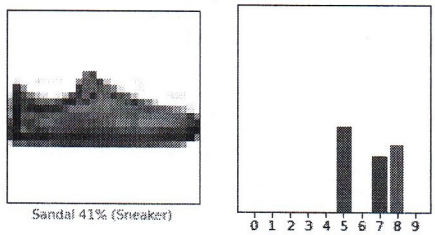
With the model trained, you can use it to make predictions about some images.

Let's look at the 0th image, predictions, and prediction array. Correct prediction labels are blue and incorrect prediction labels are red. The number gives the percentage (out of 100) for the predicted label.

In [33]:
```
i = 0
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i],  test_labels)
plt.show()
```



In [34]:
```
i = 12
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i],  test_labels)
plt.show()
```



Let's plot several images with their predictions. Note that the model can be wrong even when very confident.

In [35]:
```
# Plot the first X test images, their predicted labels, and the true labels.
# Color correct predictions in blue and incorrect predictions in red.
num_rows = 5
num_cols = 3
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], test_labels, test_images)
  plt.subplot(num_rows, 2*num_cols, 2*i+2)
  plot_value_array(i, predictions[i], test_labels)
plt.tight_layout()
plt.show()
```

Use the trained model

Finally, use the trained model to make a prediction about a single image.

In [36]:
```
# Grab an image from the test dataset.
img = test_images[1]

print(img.shape)
```

```
(28, 28)
```

`tf.keras` models are optimized to make predictions on a *batch*, or collection, of examples at once. Accordingly, even though you're using a single image, you need to add it to a list:

In [37]:
```
# Add the image to a batch where it's the only member.
img = (np.expand_dims(img,0))

print(img.shape)
```

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

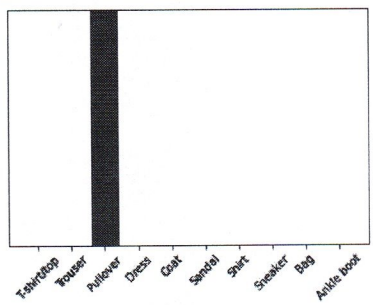
Now predict the correct label for this image:

In [38]:
```
predictions_single = probability_model.predict(img)

print(predictions_single)
```

```
[[1.7112123e-05 3.6877675e-11 9.9842179e-01 2.0823014e-09 7.8204600e-04
  4.5134244e-13 7.7893987e-04 1.5363768e-17 2.6546116e-09 8.9851412e-14]]
```

In [39]:
```
plot_value_array(1, predictions_single[0], test_labels)
_ = plt.xticks(range(10), class_names, rotation=45)
```



`tf.keras.Model.predict` returns a list of lists—one list for each image in the batch of data. Grab the predictions for our (only) image in the batch:

In [40]:
```
np.argmax(predictions_single[0])
```

Out[40]: 2

And the model predicts a label as expected.

Example: Fashion MNIST - Multi-class classification

Dataset

In [ ]:
```
# Load built-in dataset
# ------------------------------------------------------------------------------
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
```

In [ ]:
```
x_train.shape
y_train.shape
```

In [ ]:
```
x_valid = x_train[50000:, ...]
y_valid = y_train[50000:, ...]

x_train = x_train[:50000, ...]
y_train = y_train[:50000, ...]
```

In [ ]:
```
# Create Training Dataset
# ------------------------------------------------------------------------------
train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))

# Shuffle
train_dataset = train_dataset.shuffle(buffer_size=x_train.shape[0])

# Normalize images
def normalize_img(x_, y_):
    return tf.cast(x_, tf.float32) / 255., y_

train_dataset = train_dataset.map(normalize_img)

# 1-hot encoding <- for categorical cross entropy
def to_categorical(x_, y_):
    return x_, tf.one_hot(y_, depth=10)

train_dataset = train_dataset.map(to_categorical)

# Divide in batches
bs = 32
train_dataset = train_dataset.batch(bs)

# Repeat
train_dataset = train_dataset.repeat()
```

In [ ]:
```
# Create Validation Dataset
# ------------------------------------------------------------------------------
valid_dataset = tf.data.Dataset.from_tensor_slices((x_valid, y_valid))
valid_dataset = valid_dataset.map(normalize_img)
valid_dataset = valid_dataset.map(to_categorical)
valid_dataset = valid_dataset.batch(1)
valid_dataset = valid_dataset.repeat()
```

In [ ]:
```
# Create Test Dataset
# ------------------------------------------------------------------------------
test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test))
test_dataset = test_dataset.map(normalize_img)
test_dataset = test_dataset.map(to_categorical)
test_dataset = test_dataset.batch(1)
```

Model

In [ ]:
```
# Create Model
# ------------------------------------------------------------------------------
# Def Input keras tensor
x = tf.keras.Input(shape=[28, 28])

# Def intermediate hidden layers and chain
flatten = tf.keras.layers.Flatten()(x)
h = tf.keras.layers.Dense(units=10, activation=tf.keras.activations.sigmoid)(flatten)

# Def output layer and chain
out = tf.keras.layers.Dense(units=10, activation=tf.keras.activations.softmax)(h)

# Create Model instance defining inputs and outputs
model = tf.keras.Model(inputs=x, outputs=out)
```

← input layer : we're gonna accept inputs of 28x28 (actually batches of images 28x28)

← only one hidden layer with 10 units

In [ ]:
```
# Visualize created model as a table
model.summary()

# Visualize initialized weights
model.weights
```

In [ ]:
```
# Equivalent formulation (create model with sequential)
# ------------------------------------------------------------------------------
# seq_model = tf.keras.Sequential()
# seq_model.add(tf.keras.layers.Flatten(input_shape=(28, 28))) # or as a list
# seq_model.add(tf.keras.layers.Dense(units=10, activation=tf.keras.activations.sigmoid))
# seq_model.add(tf.keras.layers.Dense(units=10, activation=tf.keras.activations.softmax))
```

In [ ]:
```
# seq_model.summary()
# seq_model.weights
```

Prepare the model for training

In [ ]:
```
# Optimization params
# ----------------------------------------------------------------------------
# Loss
loss = tf.keras.losses.CategoricalCrossentropy()

# Learning rate
lr = 1e-3
optimizer = tf.keras.optimizers.Adam(learning_rate=lr)

# Validation metrics
metrics = ['accuracy']
```

= def. how the model will be trained

In [ ]:
```
# Compile Model
model.compile(optimizer=optimizer, loss=loss, metrics=metrics)
```

Training

In [ ]:
```
# Training the model
# ----------------------------------------------------------------------------
model.fit(x=train_dataset,
          y=None,
          epochs=10,
          steps_per_epoch=int(np.ceil(x_train.shape[0] / bs)),
          validation_data=valid_dataset,
          validation_steps=10000)
```