A simple RNN for fashion MNIST Classification

Dr. Leticia C. Cagnina

Mexican NLP Summer School 2021 (June 3rd, 2021)

https://ampln.github.io/escuelaverano2021/

About this notebook

The content of this notebook is based on the work published at https://github.com/codingyogini/TensorFlow-NNs/blob/master/NN_MNIST_depth.ipynb. Modifications were performed in order to adapt it to the requirements of **"Redes Neuronales: conceptos básicos y aplicaciones"** Turorial at **Mexican NLP Summer School 2021**. For doubts/suggestions, please contact me: lcagnina@gmail.com. I recommend run the notebook in Google Colab, setting the runtime type to GPU to accelerate the execution.

Content:

Exploring Neural Networks with fashion MNIST.

```
* Introducing the Fashion MNIST dataset
* Comparing different neural network by depth: 3, 6 and 12 layers (NN-3 vs NN-6 vs NN-12)
* Improving predictions with more epochs (5 vs 30)
* Visualizing predictions
```

This notebook shows a simple MultiLayer Preceptron neural network to classify fashion MNIST images. Different models will be tested in order to observe the performance of the classifier with different number of layers. Note that particular activation functions are used such as Softmax and ReLu. These are typical in the classification of categorical data.

This is an introductory practice to the topic!!! will study more interesting problems in next notebooks: Deep Learning and LSTM!!

```
import tensorflow as tf
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense
import numpy as np
import matplotlib.pyplot as plt
import math

print (tf.__version__) # 2.4.1
```

```
2.5.0
```

The fashion MNIST dataset

Zalando research published this dataset, with 10 different fashion products. Called fashion MNIST, this dataset is meant to be a replacement for the original MNIST which turned out to be too easy for machine learning folks; even linear classifiers were able to achieve high classification accuracy. This new dataset promises to be more challenging, so that machine learning algorithms have to learn more advanced features to correctly classify the images. The fashion MNIST dataset contains 70,000 greyscale images distributed in 10 categories. The images show individual articles of clothing at low resolution (28x28px). Below we can see a sample of 25 images with their 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

Loading the fashion MNIST data

```
fashion_mnist = keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
# returns 4 numpy arrays: 2 training sets and 2 test sets
# images: 28x28 arrays, pixel values: 0 to 255
# labels: array of integers: 0 to 9 => class of clothings
# Training set: 60,000 images, Testing set: 10,000 images

# class names are not included, need to create them to plot the images
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle
```

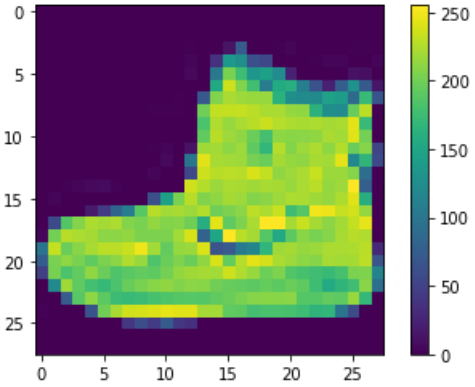
Exploring and Visualizing the data

The data can be loaded from Keras into a Training set (60,000 images) and Testing set (10,000 images). The images are 28x28 arrays with pixel values 0 to 255, and the labels are an array of integers 0 to 9, representing 10 classes of clothing. We can see the training data was stored in an array of shape (60000,28,28) and testing data in an array of shape (10000,28,28).

```
print("train_images:", train_images.shape)
print("test_images:", test_images.shape)
```

```
train_images: (60000, 28, 28)
test_images: (10000, 28, 28)
```

```
# Visualize the first image from the training dataset
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
```



Normalizing the data

The next step is to normalize the data dimensions so that they are approximately the same scale.

```
# scale the values to a range of 0 to 1 of both data sets
train_images = train_images / 255.0
test_images = test_images / 255.0
```

```
# display the first 25 images from the training set and
# display the class name below each image
# verify that data is in correct format
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]])
```

The first NN model

Step 1-Build the architecture

Step 2-Compile the model

Step 3-Train the model

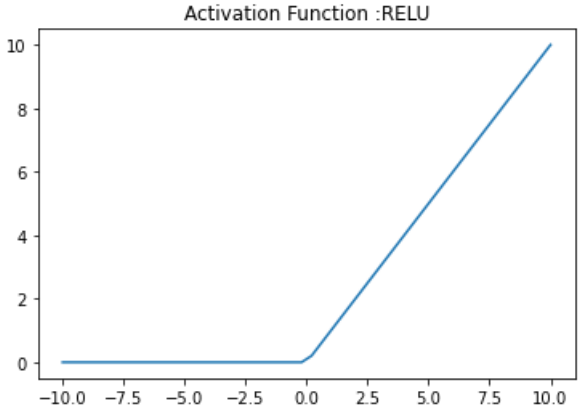
Step 4-Evaluate the model

Step1: The arquitecture

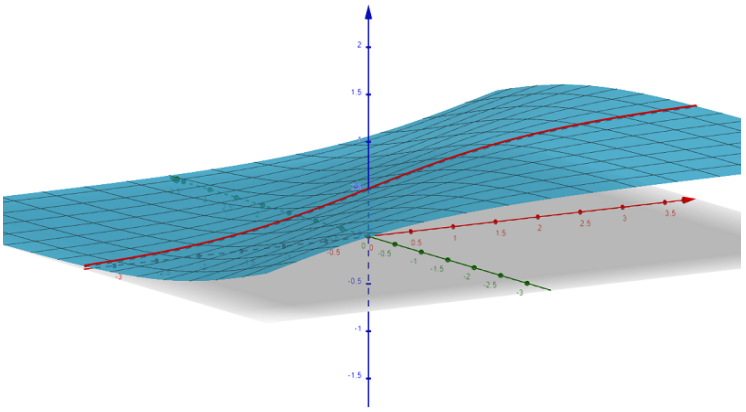
First, we'll design the NN architecture by deciding the number of layers and activation functions. We'll start with a simple 3-layer Neural Network. In the first layer we 'flatten' the data, so that a (28x28) shape flattens to 784. The second layer is a

dense layer with a ReLu activation function and has 128 neurons. The last layer is a dense layer with a softmax activation function that classifies the 10 categories of the data and has 10 neurons.

- Flatten layer: only flattens the input.
- Dense layer: regular densely-connected NN layer. Dense implements the operation: output = activation(dot(input, weight) + bias)
- ReLu activation function: computes rectified linear: max(input, 0). It has become the default activation function for many types of neural networks because a model that uses it is easier to train and often achieves better performance. Overcomes the vanishing gradient problem (the model stops learning), allowing models to learn faster and perform better. It is the default activation when developing multilayer Perceptron and convolutional neural networks. Because the rectified function is linear for half of the input domain and nonlinear for the other half, it is referred to as a piecewise linear function. Mathematically, it is written as $f(x) = max(0, x)$ and it looks like:



- Softmax activation function: converts a real vector (the output) to a vector of categorical probabilities. The elements of the output vector are in range (0, 1) and sum to 1. Softmax is often used as the activation for the last layer of a classification network because the result could be interpreted as a probability distribution. The softmax of each vector $\mathbf{x}$ is computed as $f(\mathbf{x}) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$ for each element i, j of $\mathbf{x}$. Many multi-layer neural networks end in a penultimate layer which outputs real-valued scores that are not conveniently scaled and which may be difficult to work with. Here the softmax is very useful because it converts the scores to a normalized probability distribution, which can be displayed to the user. For this reason it is usual to append a softmax function as the final layer of the neural network. For a 2-dim vector, softmax looks like (https://themaverickmeerkat.com/2019-10-23-Softmax/):



- The model summary table provides a nice visualization of the network architecture and parameters.

```
# Step 1 - Build the architecture
# Model: a simple 3-layer neural network
model_3 = keras.Sequential([
    keras.layers.Flatten(input_shape=(28,28)),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(10, activation=tf.nn.softmax)
```

```
])
model_3.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
_________________________________________________________________
```

Step 2: the model

`compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])` : configures the model for the training, using `optimizer` with the objective function `loss` and the results will be reported in base to `metrics`.

Loss function — calculates the difference between the output and the target variable. It measures the accuracy of the model during training and we want to minimize this function.

Optimizer — how the model is updated and is based on the data and the loss function.

Metrics — monitors the training and testing steps. Accuracy is a common metric and it measures the fraction of images that are correctly classified.

- Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments. According to Kingma et al., 2014, the method is "computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters". Also is effective.
- `sparse_categorical_crossentropy` computes the crossentropy loss between the labels and predictions. This crossentropy loss function is used when there are two or more label classes. It expects labels as integers (no one-hot representation).

```
# Step 2 - Compile the model
model_3.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Step 3: Training

We train the model by fitting it to the training data, so we give it the input (images) and expected output (labels). Here, an important step to minimize overfitting is validation. There are a few ways to validate, in this case, we use the automatic validation built into the function, where we set the validation_split on the training data. Here we use an 80/20 split: 80% for training and 20% for validating. We also need to define how many times the network will be trained, this is an `epoch`. It's an *arbitrary* cutoff and here we choose 5 epochs. Take into account:

Epoch — one forward pass and one backward pass of *all* the training examples

Iteration — number of passes, (forward and backward) to complete one epoch

Example: if you have 1,000 training examples, and your batch size is 500, then it will take 2 iterations to complete 1 epoch.

Training will show you the following results per epoch, note that with each epoch, the loss decreases and the accuracy increases, meaning our model is improving.

```
#Step 3 - Train the model, by fitting it to the training data
```

```
# 5 epochs, and split the training set into 80/20 for validation
# batch size = default=32
#steps_per_epoch: by default is num_samples_train/batch_size then 1500 in this case (48000/32) --> 1500 times pas

model_3.fit(train_images, train_labels, epochs=5, validation_split=0.2,shuffle=True)
```

```
Epoch 1/5
1500/1500 [==============================] - 5s 3ms/step - loss: 0.5197 - accuracy: 0.8175 - val_loss: 0.433
Epoch 2/5
1500/1500 [==============================] - 4s 3ms/step - loss: 0.3894 - accuracy: 0.8604 - val_loss: 0.373
Epoch 3/5
1500/1500 [==============================] - 4s 3ms/step - loss: 0.3447 - accuracy: 0.8747 - val_loss: 0.376
Epoch 4/5
1500/1500 [==============================] - 4s 3ms/step - loss: 0.3201 - accuracy: 0.8825 - val_loss: 0.339
Epoch 5/5
1500/1500 [==============================] - 4s 3ms/step - loss: 0.3001 - accuracy: 0.8892 - val_loss: 0.340
<tensorflow.python.keras.callbacks.History at 0x7f2830600d10>
```

Step 4: the evaluation of the model on Test set

Now that we've set up and trained our model, we need to evaluate its performance. This is done on a test dataset, new data that the model hasn't seen yet. We have to make sure to separate our training and validating dataset from our testing dataset.

```
#Step 4 - Evaluate the model
test_loss, test_acc = model_3.evaluate(test_images, test_labels)
print("Model - 3 layers - test loss: {:.2f}%".format(test_loss * 100))
print("Model - 3 layers - test accuracy: {:.2f}%".format(test_acc * 100))
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.3649 - accuracy: 0.8692
Model - 3 layers - test loss: 36.49%
Model - 3 layers - test accuracy: 86.92%
```

Making predictions with the NN-3 model trained for 5 epochs

```
# confidence of the model that the image corresponds to the label
predictions = model_3.predict(test_images)
predictions.shape #(10000, 10)
predictions[0]
```

```
array([1.3524811e-06, 9.1571666e-09, 1.8794503e-06, 1.0869107e-06,
       9.1999752e-08, 1.8472365e-03, 1.1023646e-06, 1.3455097e-02,
       1.8354936e-05, 9.8467386e-01], dtype=float32)
```

```
np.argmax(predictions[0])
```

```
9
```

```
class_names[9]
```

```
'Ankle boot'
```

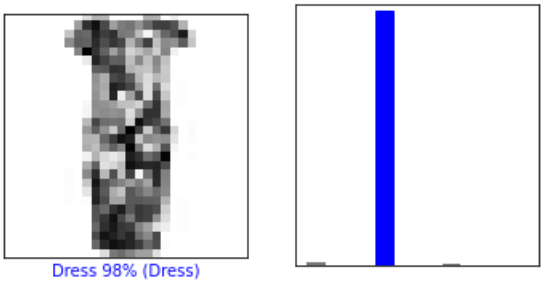
```
#Ankle boot has the highest confidence value
test_labels[0]
```

```
9
```

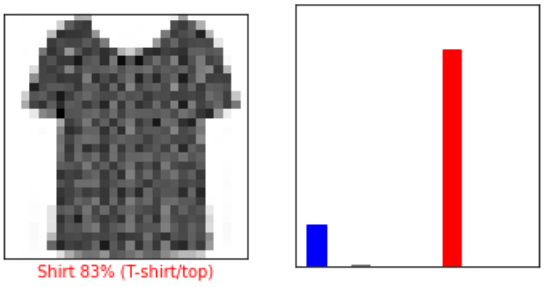
```
# plot image in a grid
def plot_image(i, predictions_array, true_label, img):
  predictions_array, true_label, img = predictions_array[i], true_label[i], img[i]
  plt.grid(False)
```

```
  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)
# plot the value array
def plot_value_array(i, predictions_array, true_label):
    predictions_array, true_label = predictions_array[i], true_label[i]
    plt.grid(False)
    plt.xticks([])
    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')
```

```
# look at 100th image, predictions, prediction array
i=100
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions, test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions, test_labels)
```

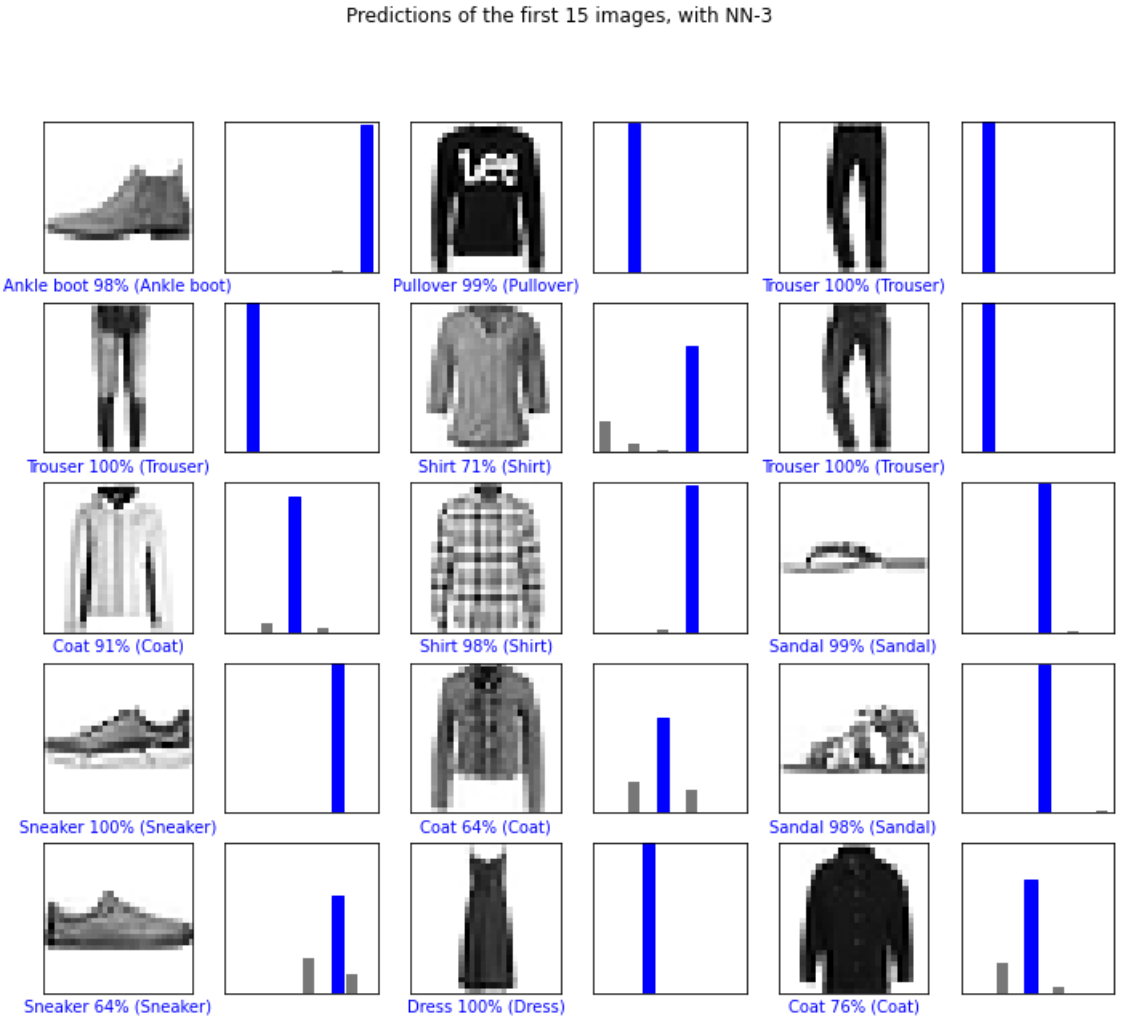



```
i = 1000
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions, test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions,  test_labels)
```




```
# Plot the first 15 test images, their predicted label, and the true label
# Color correct predictions in blue, 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))
plt.suptitle("Predictions of the first 15 images, with NN-3")
for i in range(num_images):
  plt.subplot(num_rows, 2*num_cols, 2*i+1)
  plot_image(i, predictions, test_labels, test_images)
  plt.subplot(num_rows, 2*num_cols, 2*i+2)
  plot_value_array(i, predictions, test_labels)
```



Is deeper more accurate?

Comparing network depth, with a NN-6 and NN-12

```
# Model a simple 6-layer neural network
model_6 = keras.Sequential([
    keras.layers.Flatten(input_shape=(28,28)),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(10, activation=tf.nn.softmax)
])
#model_6.summary()
model_6.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

```
#Train the NN-6 with 5 epochs
```

```
model_6.fit(train_images, train_labels, epochs=5, validation_split=0.2)

#Evaluate the model with test datasets
test_loss, test_acc = model_6.evaluate(test_images, test_labels)
print("Model - 6 layers - test loss: {:.2f}%".format(test_loss * 100))
print("Model - 6 layers - test accuracy: {:.2f}%".format(test_acc * 100))
```

```
Epoch 1/5
1500/1500 [==============================] - 5s 3ms/step - loss: 0.5219 - accuracy: 0.8104 - val_loss: 0.418
Epoch 2/5
1500/1500 [==============================] - 5s 3ms/step - loss: 0.3818 - accuracy: 0.8602 - val_loss: 0.397
Epoch 3/5
1500/1500 [==============================] - 5s 3ms/step - loss: 0.3457 - accuracy: 0.8736 - val_loss: 0.360
Epoch 4/5
1500/1500 [==============================] - 5s 3ms/step - loss: 0.3201 - accuracy: 0.8823 - val_loss: 0.350
Epoch 5/5
1500/1500 [==============================] - 5s 3ms/step - loss: 0.3056 - accuracy: 0.8860 - val_loss: 0.337
313/313 [==============================] - 1s 3ms/step - loss: 0.3626 - accuracy: 0.8706
Model - 6 layers - test loss: 36.26%
Model - 6 layers - test accuracy: 87.06%
```

```
# Model a simple 12-layer neural network
model_12 = keras.Sequential([
    keras.layers.Flatten(input_shape=(28,28)),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(10, activation=tf.nn.softmax)
])
#model_12.summary()
model_12.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

```
#Train the NN-12 with 5 epochs
model_12.fit(train_images, train_labels, epochs=5, validation_split=0.2)

#Evaluate the model
test_loss, test_acc = model_12.evaluate(test_images, test_labels)
print("Model - 12 layers - test loss: {:.2f}%".format(test_loss * 100))
print("Model - 12 layers - test accuracy: {:.2f}%".format(test_acc * 100))
```

```
Epoch 1/5
1500/1500 [==============================] - 7s 5ms/step - loss: 0.6399 - accuracy: 0.7577 - val_loss: 0.554
Epoch 2/5
1500/1500 [==============================] - 6s 4ms/step - loss: 0.4492 - accuracy: 0.8396 - val_loss: 0.411
Epoch 3/5
1500/1500 [==============================] - 7s 4ms/step - loss: 0.3998 - accuracy: 0.8563 - val_loss: 0.403
Epoch 4/5
1500/1500 [==============================] - 7s 4ms/step - loss: 0.3767 - accuracy: 0.8664 - val_loss: 0.407
Epoch 5/5
1500/1500 [==============================] - 6s 4ms/step - loss: 0.3580 - accuracy: 0.8719 - val_loss: 0.369
313/313 [==============================] - 1s 3ms/step - loss: 0.3915 - accuracy: 0.8667
Model - 12 layers - test loss: 39.15%
Model - 12 layers - test accuracy: 86.67%
```

Does increasing epochs improve our classification?

▾ Re-train the NN-3 with 30 epochs, and plot the loss and accuracy

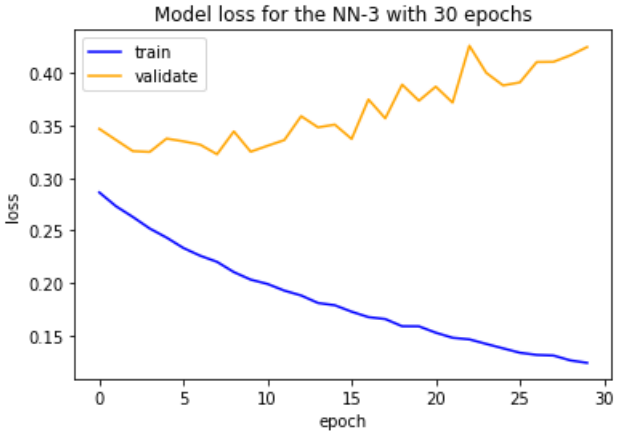
```
# NN-3, 30 epochs
history_NN3_30=model_3.fit(train_images, train_labels, epochs=30, validation_split=0.2)

test_loss, test_acc = model_3.evaluate(test_images, test_labels)
print("Model 30 - 3 layers - test loss: {:.2f}%".format(test_loss * 100))
print("Model 30 - 3 layers - test accuracy: {:.2f}%".format(test_acc * 100))
```

```
1500/1500 [==============================] - 4s 3ms/step - loss: 0.2629 - accuracy: 0.9019 - val_loss: 0
Epoch 4/30
1500/1500 [==============================] - 4s 2ms/step - loss: 0.2519 - accuracy: 0.9065 - val_loss: 0
Epoch 5/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.2433 - accuracy: 0.9096 - val_loss: 0
Epoch 6/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.2333 - accuracy: 0.9137 - val_loss: 0
Epoch 7/30
1500/1500 [==============================] - 4s 2ms/step - loss: 0.2260 - accuracy: 0.9156 - val_loss: 0
Epoch 8/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.2202 - accuracy: 0.9175 - val_loss: 0
Epoch 9/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.2104 - accuracy: 0.9221 - val_loss: 0
Epoch 10/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.2032 - accuracy: 0.9237 - val_loss: 0
Epoch 11/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.1991 - accuracy: 0.9261 - val_loss: 0
Epoch 12/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.1928 - accuracy: 0.9285 - val_loss: 0
Epoch 13/30
1500/1500 [==============================] - 4s 2ms/step - loss: 0.1881 - accuracy: 0.9308 - val_loss: 0
Epoch 14/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.1810 - accuracy: 0.9326 - val_loss: 0
Epoch 15/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.1788 - accuracy: 0.9320 - val_loss: 0
Epoch 16/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.1727 - accuracy: 0.9350 - val_loss: 0
Epoch 17/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.1675 - accuracy: 0.9386 - val_loss: 0
Epoch 18/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.1657 - accuracy: 0.9379 - val_loss: 0
Epoch 19/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.1588 - accuracy: 0.9404 - val_loss: 0
Epoch 20/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.1587 - accuracy: 0.9407 - val_loss: 0
Epoch 21/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.1527 - accuracy: 0.9425 - val_loss: 0
Epoch 22/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.1478 - accuracy: 0.9445 - val_loss: 0
Epoch 23/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.1463 - accuracy: 0.9452 - val_loss: 0
Epoch 24/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.1420 - accuracy: 0.9468 - val_loss: 0
Epoch 25/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.1378 - accuracy: 0.9480 - val_loss: 0
Epoch 26/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.1336 - accuracy: 0.9506 - val_loss: 0
Epoch 27/30
1500/1500 [==============================] - 4s 2ms/step - loss: 0.1315 - accuracy: 0.9502 - val_loss: 0
Epoch 28/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.1309 - accuracy: 0.9512 - val_loss: 0
Epoch 29/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.1263 - accuracy: 0.9515 - val_loss: 0
Epoch 30/30
1500/1500 [==============================] - 4s 3ms/step - loss: 0.1240 - accuracy: 0.9534 - val_loss: 0
313/313 [==============================] - 1s 3ms/step - loss: 0.4663 - accuracy: 0.8816
Model 30 - 3 layers - test loss: 46.63%
Model 30 - 3 layers - test accuracy: 88.16%
```

```
#Plot loss results for training data and testing data
plt.plot(history_NN3_30.history['loss'], 'blue')
plt.plot(history_NN3_30.history['val_loss'], 'orange')
```

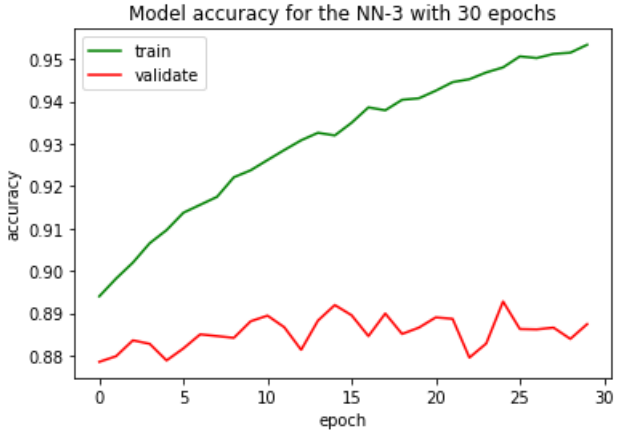
```
plt.title('Model loss for the NN-3 with 30 epochs')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validate'], loc='upper left')
```

```
<matplotlib.legend.Legend at 0x7f27f16cf850>
```




```
#Plot accuracy results for training data and testing data
plt.plot(history_NN3_30.history['accuracy'], 'green')
plt.plot(history_NN3_30.history['val_accuracy'], 'red')
plt.title('Model accuracy for the NN-3 with 30 epochs')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validate'], loc='upper left')
```

```
<matplotlib.legend.Legend at 0x7f27f16ed750>
```



Re-train the NN-6 with 30 epochs, and plot the loss and accuracy

```
# NN-6, 30 epochs
history_NN6_30=model_6.fit(train_images, train_labels, epochs=30, validation_split=0.2)

test_loss, test_acc = model_6.evaluate(test_images, test_labels)
print("Model 30 - 6 layers - test loss: {:.2f}%".format(test_loss * 100))
print("Model 30 - 6 layers - test accuracy: {:.2f}%".format(test_acc * 100))
```

```
Epoch 4/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.2531 - accuracy: 0.9047 - val_loss: 0
Epoch 5/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.2493 - accuracy: 0.9056 - val_loss: 0

Epoch 6/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.2369 - accuracy: 0.9098 - val_loss: 0
Epoch 7/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.2283 - accuracy: 0.9136 - val_loss: 0
Epoch 8/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.2248 - accuracy: 0.9146 - val_loss: 0
Epoch 9/30
```

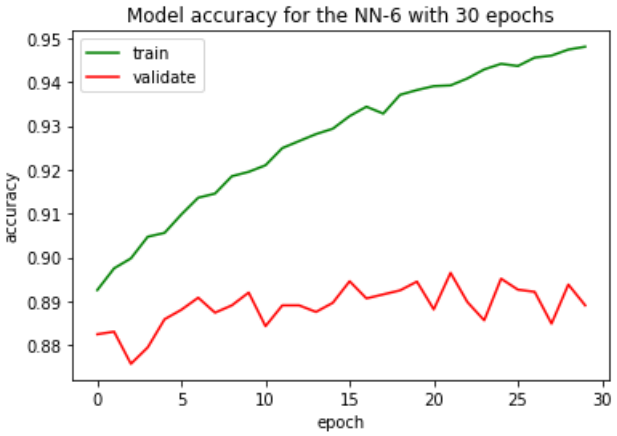
```
Epoch 9/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.2142 - accuracy: 0.9185 - val_loss: 0
Epoch 10/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.2113 - accuracy: 0.9195 - val_loss: 0
Epoch 11/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.2070 - accuracy: 0.9210 - val_loss: 0
Epoch 12/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1964 - accuracy: 0.9249 - val_loss: 0
Epoch 13/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1918 - accuracy: 0.9265 - val_loss: 0
Epoch 14/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1884 - accuracy: 0.9281 - val_loss: 0
Epoch 15/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1828 - accuracy: 0.9294 - val_loss: 0
Epoch 16/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1800 - accuracy: 0.9322 - val_loss: 0
Epoch 17/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1705 - accuracy: 0.9344 - val_loss: 0
Epoch 18/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1738 - accuracy: 0.9328 - val_loss: 0
Epoch 19/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1659 - accuracy: 0.9371 - val_loss: 0
Epoch 20/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1632 - accuracy: 0.9382 - val_loss: 0
Epoch 21/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1587 - accuracy: 0.9391 - val_loss: 0
Epoch 22/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1572 - accuracy: 0.9392 - val_loss: 0
Epoch 23/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1561 - accuracy: 0.9408 - val_loss: 0
Epoch 24/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1512 - accuracy: 0.9429 - val_loss: 0
Epoch 25/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1479 - accuracy: 0.9441 - val_loss: 0
Epoch 26/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1454 - accuracy: 0.9436 - val_loss: 0
Epoch 27/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1417 - accuracy: 0.9456 - val_loss: 0
Epoch 28/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1404 - accuracy: 0.9460 - val_loss: 0
Epoch 29/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1383 - accuracy: 0.9474 - val_loss: 0
Epoch 30/30
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1341 - accuracy: 0.9480 - val_loss: 0
313/313 [==============================] - 1s 3ms/step - loss: 0.5723 - accuracy: 0.8752
Model 30 - 6 layers - test loss: 57.23%
Model 30 - 6 layers - test accuracy: 87.52%
```

```
#Plot loss results for training data and testing data
plt.plot(history_NN6_30.history['loss'], 'blue')
plt.plot(history_NN6_30.history['val_loss'], 'orange')
plt.title('Model loss for the NN-6 with 30 epochs')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validate'], loc='upper left')
```

```
<matplotlib.legend.Legend at 0x7f27f1560b10>
```

```
#Plot accuracy results for training data and testing data
plt.plot(history_NN6_30.history['accuracy'], 'green')
plt.plot(history_NN6_30.history['val_accuracy'],'red')
plt.title('Model accuracy for the NN-6 with 30 epochs')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validate'], loc='upper left')
```

```
<matplotlib.legend.Legend at 0x7f27f15e6fd0>
```



Re-train the NN-12 with 30 epochs, and plot the loss and accuracy

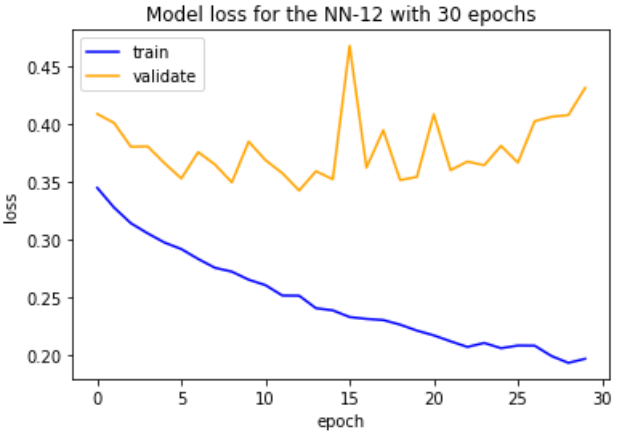
```
# NN-12, 30 epochs
history_NN12_30=model_12.fit(train_images, train_labels, epochs=30, validation_split=0.2)

test_loss, test_acc = model_12.evaluate(test_images, test_labels)
print("Model 30 - 12 layers - test loss: {:.2f}%".format(test_loss * 100))
print("Model 30 - 12 layers - test accuracy: {:.2f}%".format(test_acc * 100))
```

```
Epoch 4/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.3052 - accuracy: 0.8903 - val_loss: 0
Epoch 5/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2973 - accuracy: 0.8926 - val_loss: 0
Epoch 6/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2917 - accuracy: 0.8942 - val_loss: 0
Epoch 7/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2831 - accuracy: 0.8980 - val_loss: 0
Epoch 8/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2755 - accuracy: 0.9002 - val_loss: 0
Epoch 9/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2722 - accuracy: 0.9024 - val_loss: 0
Epoch 10/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2653 - accuracy: 0.9039 - val_loss: 0
Epoch 11/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2605 - accuracy: 0.9060 - val_loss: 0
Epoch 12/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2516 - accuracy: 0.9087 - val_loss: 0
Epoch 13/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2515 - accuracy: 0.9088 - val_loss: 0
Epoch 14/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2406 - accuracy: 0.9128 - val_loss: 0
Epoch 15/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2388 - accuracy: 0.9146 - val_loss: 0
Epoch 16/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2330 - accuracy: 0.9150 - val_loss: 0
Epoch 17/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2314 - accuracy: 0.9159 - val_loss: 0
Epoch 18/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2304 - accuracy: 0.9154 - val_loss: 0
Epoch 19/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2265 - accuracy: 0.9184 - val_loss: 0
Epoch 20/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2213 - accuracy: 0.9195 - val_loss: 0
```

```
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2213 - accuracy: 0.9195 - val_loss: 0
Epoch 21/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2172 - accuracy: 0.9202 - val_loss: 0
Epoch 22/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2121 - accuracy: 0.9244 - val_loss: 0
Epoch 23/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2072 - accuracy: 0.9231 - val_loss: 0
Epoch 24/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2107 - accuracy: 0.9233 - val_loss: 0
Epoch 25/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2061 - accuracy: 0.9253 - val_loss: 0
Epoch 26/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2084 - accuracy: 0.9247 - val_loss: 0
Epoch 27/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.2083 - accuracy: 0.9248 - val_loss: 0
Epoch 28/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.1994 - accuracy: 0.9272 - val_loss: 0
Epoch 29/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.1935 - accuracy: 0.9290 - val_loss: 0
Epoch 30/30
1500/1500 [==============================] - 7s 4ms/step - loss: 0.1970 - accuracy: 0.9275 - val_loss: 0
313/313 [==============================] - 1s 3ms/step - loss: 0.4322 - accuracy: 0.8804
Model 30 - 12 layers - test loss: 43.22%
Model 30 - 12 layers - test accuracy: 88.04%
```

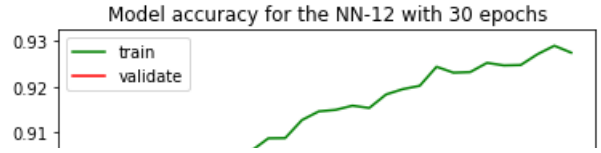
```
#Plot loss results for training data and testing data
plt.plot(history_NN12_30.history['loss'], 'blue')
plt.plot(history_NN12_30.history['val_loss'], 'orange')
plt.title('Model loss for the NN-12 with 30 epochs')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validate'], loc='upper left')
```

```
<matplotlib.legend.Legend at 0x7f27f147ca50>
```




```
#Plot accuracy results for training data and testing data
plt.plot(history_NN12_30.history['accuracy'], 'green')
plt.plot(history_NN12_30.history['val_accuracy'], 'red')
plt.title('Model accuracy for the NN-12 with 30 epochs')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validate'], loc='upper left')
```

```
<matplotlib.legend.Legend at 0x7f27f154c990>
```



Making predictions with the NN-3 model trained for 30 epochs

```
# confidence of the model that the image corresponds to the label
predictions = model_3.predict(test_images)
predictions.shape #(10000, 10)
predictions[0]
```

```
array([1.1828169e-15, 3.2970135e-18, 2.1225974e-11, 1.0823418e-15,
       2.2704120e-13, 7.2748589e-09, 8.6054525e-11, 1.4620359e-04,
       3.8369603e-13, 9.9985373e-01], dtype=float32)
```

```
np.argmax(predictions[0])
```

```
9
```

```
class_names[9]
```

```
'Ankle boot'
```

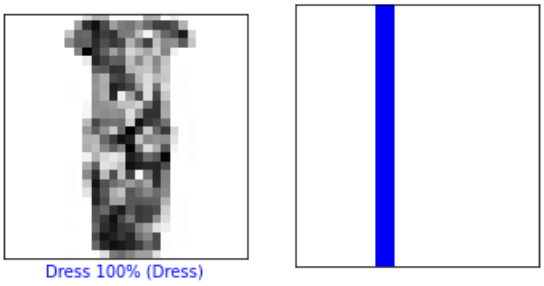
```
#Ankle boot has the highest confidence value
test_labels[0]
```

```
9
```

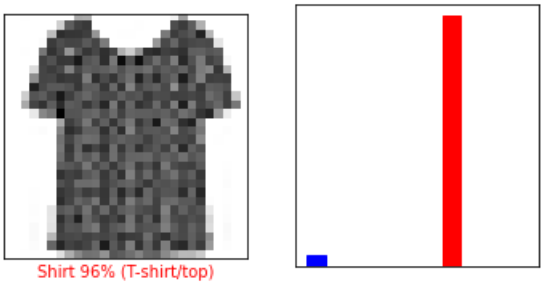
```
# plot image in a grid
def plot_image(i, predictions_array, true_label, img):
  predictions_array, true_label, img = predictions_array[i], true_label[i], img[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])

  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)
# plot the value array
def plot_value_array(i, predictions_array, true_label):
    predictions_array, true_label = predictions_array[i], true_label[i]
    plt.grid(False)
    plt.xticks([])
    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')
```

```
# look at 100th image, predictions, prediction array
```

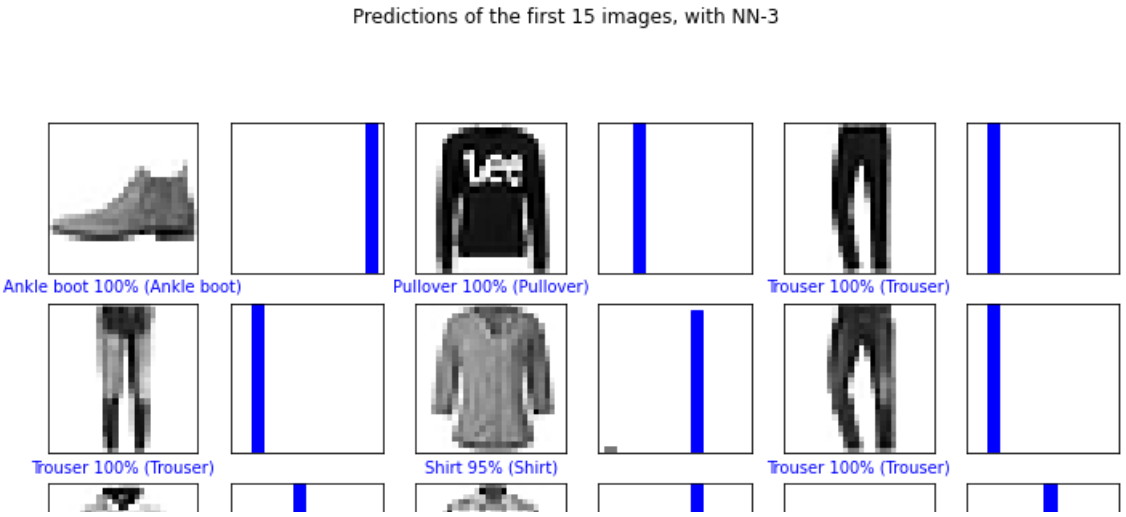
```
i=100
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions, test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions, test_labels)
```




```
i = 1000
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions, test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions,  test_labels)
```




```
# Plot the first 15 test images, their predicted label, and the true label
# Color correct predictions in blue, 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))
plt.suptitle("Predictions of the first 15 images, with NN-3")
for i in range(num_images):
  plt.subplot(num_rows, 2*num_cols, 2*i+1)
  plot_image(i, predictions, test_labels, test_images)
  plt.subplot(num_rows, 2*num_cols, 2*i+2)
  plot_value_array(i, predictions, test_labels)
```



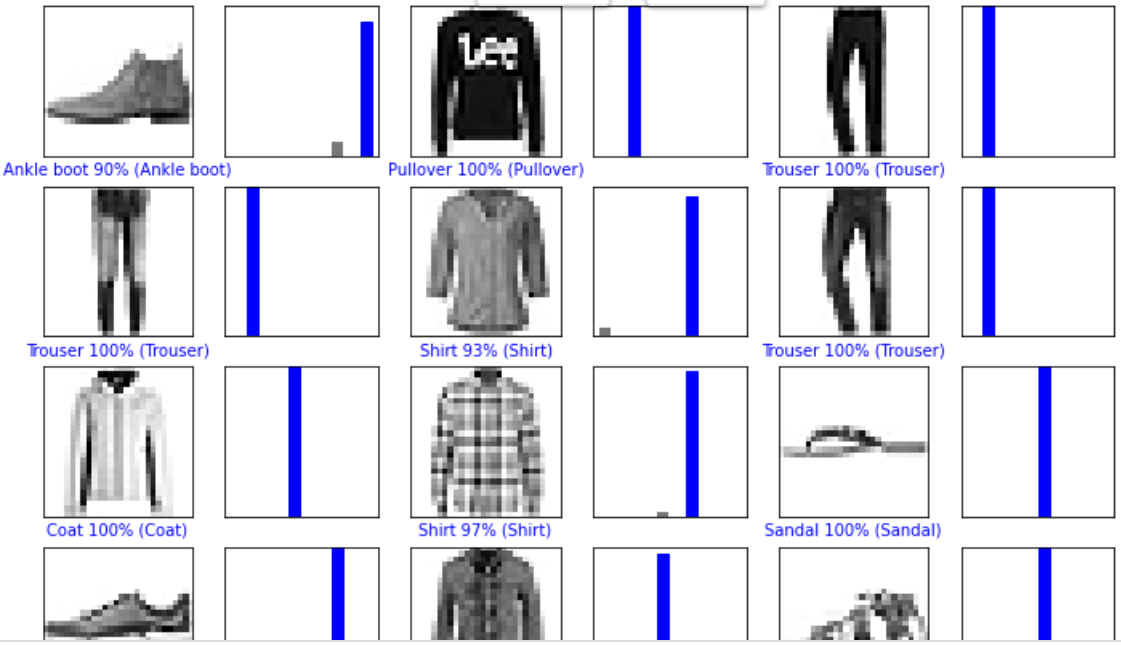
Making predictions with the NN-12

```
predictions = model_12.predict(test_images)
```

```
# Plot the first 15 test images, their predicted label, and the true label
# Color correct predictions in blue, 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))
plt.suptitle("Predictions of first 15 images, with NN-12")
for i in range(num_images):
  plt.subplot(num_rows, 2*num_cols, 2*i+1)
  plot_image(i, predictions, test_labels, test_images)
  plt.subplot(num_rows, 2*num_cols, 2*i+2)
  plot_value_array(i, predictions, test_labels)
```

Predictions of first 15 images, with NN-12

+ Código + Texto



✓ 2 s se ejecutó 10:14