Programming Assignment

Residual network

Instructions

In this notebook, you will use the model subclassing API together with custom layers to create a residual network architecture. You will then train your custom model on the Fashion-MNIST dataset by using a custom training loop and implementing the automatic differentiation tools in Tensorflow to calculate the gradients for backpropagation.

Some code cells are provided you in the notebook. You should avoid editing provided code, and make sure to execute the cells in order to avoid unexpected errors. Some cells begin with the line:

`#### GRADED CELL ####`

Don't move or edit this first line - this is what the automatic grader looks for to recognise graded cells. These cells require you to write your own code to complete them, and are automatically graded when you submit the notebook. Don't edit the function name or signature provided in these cells, otherwise the automatic grader might not function properly. Inside these graded cells, you can use any functions or classes that are imported below, but make sure you don't use any variables that are outside the scope of the function.

How to submit

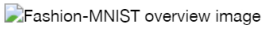
Complete all the tasks you are asked for in the worksheet. When you have finished and are happy with your code, press the **Submit Assignment** button at the top of this notebook.

Let's get started!

We'll start running some imports, and loading the dataset. Do not edit the existing imports in the following cell. If you would like to make further Tensorflow imports, you should add them here.

In [1]:
```
#### PACKAGE IMPORTS ####

# Run this cell first to import all required packages. Do not make any imports elsewhere in the notebook

import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Layer, BatchNormalization, Conv2D, Dense, Flatten, Add
import numpy as np
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt

# If you would like to make further imports from tensorflow, add them here

from tensorflow.keras.layers import Layer, BatchNormalization, Conv2D, Dense, Flatten, Add, Softmax
```

executed in 12.0s, finished 13:57:56 2021-03-25



The Fashion-MNIST dataset

In this assignment, you will use the Fashion-MNIST dataset. It consists of a training set of 60,000 images of fashion items with corresponding labels, and a test set of 10,000 images. The images have been normalised and centred. The dataset is frequently used in machine learning research, especially as a drop-in replacement for the MNIST dataset.

- H. Xiao, K. Rasul, and R. Vollgraf. "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms." arXiv:1708.07747, August 2017.

Your goal is to construct a ResNet model that classifies images of fashion items into one of 10 classes.

Load the dataset

For this programming assignment, we will take a smaller sample of the dataset to reduce the training time.

In [2]:
```
# Load and preprocess the Fashion-MNIST dataset

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

train_images = train_images.astype(np.float32)
test_images = test_images.astype(np.float32)

train_images = train_images[:5000] / 255.
train_labels = train_labels[:5000]

test_images = test_images / 255.

train_images = train_images[..., np.newaxis]
test_images = test_images[..., np.newaxis]
```

executed in 590ms, finished 13:58:00 2021-03-25

In [3]:
```
# Create Dataset objects for the training and test sets

train_dataset = tf.data.Dataset.from_tensor_slices((train_images, train_labels))
train_dataset = train_dataset.batch(32)

test_dataset = tf.data.Dataset.from_tensor_slices((test_images, test_labels))
test_dataset = test_dataset.batch(32)
```

executed in 823ms, finished 13:58:02 2021-03-25

In [4]:
```
# Get dataset labels

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

executed in 5ms, finished 13:58:05 2021-03-25

Create custom layers for the residual blocks

You should now create a first custom layer for a residual block of your network. Using layer subclassing, build your custom layer according to the following spec:

- The custom layer class should have `__init__`, `build` and `call` methods. The `__init__` method has been completed for you. It calls the base `Layer` class initializer, passing on any keyword arguments
- The `build` method should create the layers. It will take an `input_shape` argument, and should extract the number of filters from this argument. It should create:
  - A BatchNormalization layer: this will be the first layer in the block, so should use its `input shape` keyword argument
  - A Conv2D layer with the same number of filters as the layer input, a 3x3 kernel size, `'SAME'` padding, and no activation function
  - Another BatchNormalization layer
  - Another Conv2D layer, again with the same number of filters as the layer input, a 3x3 kernel size, `'SAME'` padding, and no activation function
- The `call` method should then process the input through the layers:
  - The first BatchNormalization layer: ensure to set the `training` keyword argument
  - A `tf.nn.relu` activation function
  - The first Conv2D layer
  - The second BatchNormalization layer: ensure to set the `training` keyword argument
  - Another `tf.nn.relu` activation function
  - The second Conv2D layer
  - It should then add the layer inputs to the output of the second Conv2D layer. This is the final layer output

In [5]:
```
#### GRADED CELL ####

# Complete the following class.
# Make sure to not change the class or method names or arguments.

class ResidualBlock(Layer):

    def __init__(self, **kwargs):


    def build(self, input_shape):
        """
        This method should build the layers according to the above specification. Make sure
        to use the input_shape argument to get the correct number of filters, and to set the
        input_shape of the first layer in the block.
        """



    def call(self, inputs, training=False):
        """
        This method should contain the code for calling the layer according to the above
        specification, using the layer objects set up in the build method.
        """

```

executed in 8ms, finished 13:58:09 2021-03-25

In [6]:
```
# Test your custom layer - the following should create a model using your layer

test_model = tf.keras.Sequential([ResidualBlock(input_shape=(28, 28, 1), name="residual_block")])
test_model.summary()
```

executed in 145ms, finished 13:58:10 2021-03-25

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
residual_block (ResidualBloc (None, 28, 28, 1)         28
=================================================================
Total params: 28
Trainable params: 24
Non-trainable params: 4
_________________________________________________________________
```

You should now create a second custom layer for a residual block of your network. This layer will be used to change the number of filters within the block. Using layer subclassing, build your custom layer according to the following spec:

- The custom layer class should have `__init__`, `build` and `call` methods
- The class initialiser should call the base `Layer` class initializer, passing on any keyword arguments. It should also accept a `out_filters` argument, and save it as a class attribute
- The `build` method should create the layers. It will take an `input_shape` argument, and should extract the number of input filters from this argument. It should create:
  - A BatchNormalization layer: this will be the first layer in the block, so should use its `input shape` keyword argument

- A Conv2D layer with the same number of filters as the layer input, a 3x3 kernel size, `"SAME"` padding, and no activation function
- Another BatchNormalization layer
- Another Conv2D layer with `out_filters` number of filters, a 3x3 kernel size, `"SAME"` padding, and no activation function
- A final Conv2D layer with `out_filters` number of filters, a 1x1 kernel size, and no activation function

- The `call` method should then process the input through the layers:
  - The first BatchNormalization layer: ensure to set the `training` keyword argument
  - A `tf.nn.relu` activation function
  - The first Conv2D layer
  - The second BatchNormalization layer: ensure to set the `training` keyword argument
  - Another `tf.nn.relu` activation function
  - The second Conv2D layer
  - It should then take the layer inputs, pass it through the final 1x1 Conv2D layer, and add to the output of the second Conv2D layer. This is the final layer output

In [7]:
```
#### GRADED CELL ####

# Complete the following class.
# Make sure to not change the class or method names or arguments.

class FiltersChangeResidualBlock(Layer):

    def __init__(self, out_filters, **kwargs):
        """
        The class initialiser should call the base class initialiser, passing any keyword
        arguments along. It should also set the number of filters as a class attribute.
        """


    def build(self, input_shape):
        """
        This method should build the layers according to the above specification. Make sure
        to use the input_shape argument to get the correct number of filters, and to set the
        input_shape of the first layer in the block.
        """

    def call(self, inputs, training=False):
        """
        This method should contain the code for calling the layer according to the above
        specification, using the layer objects set up in the build method.
        """

```

executed in 8ms, finished 13:58:17 2021-03-25

In [8]:
```
# Test your custom layer - the following should create a model using your layer

test_model = tf.keras.Sequential([FiltersChangeResidualBlock(16, input_shape=(32, 32, 3), name="fc_resnet_block")])
test_model.summary()
```

executed in 112ms, finished 13:58:18 2021-03-25

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
fc_resnet_block (FiltersChan (None, 32, 32, 16)        620
=================================================================
Total params: 620
Trainable params: 608
Non-trainable params: 12
_________________________________________________________________
```

Create a custom model that integrates the residual blocks

You are now ready to build your ResNet model. Using model subclassing, build your model according to the following spec:

- The custom model class should have `__init__` and `call` methods.
- The class initialiser should call the base `Model` class initializer, passing on any keyword arguments. It should create the model layers:
  - The first Conv2D layer, with 32 filters, a 7x7 kernel and stride of 2.
  - A `ResidualBlock` layer.
  - The second Conv2D layer, with 32 filters, a 3x3 kernel and stride of 2.
  - A `FiltersChangeResidualBlock` layer, with 64 output filters.
  - A Flatten layer
  - A final Dense layer, with a 10-way softmax output
- The `call` method should then process the input through the layers in the order given above. Ensure to pass the `training` keyword argument to the residual blocks, to ensure the correct mode of operation for the batch norm layers.

In total, your neural network should have six layers (counting each residual block as one layer).

In [9]:
```
#### GRADED CELL ####

# Complete the following class.
# Make sure to not change the class or method names or arguments.

class ResNetModel(Model):

    def __init__(self, **kwargs):
        """
        The class initialiser should call the base class initialiser, passing any keyword
        arguments along. It should also create the layers of the network according to the
        above specification.
        """

```

```
    def call(self, inputs, training=False):
        """
        This method should contain the code for calling the layer according to the above
        specification, using the layer objects set up in the initialiser.
        """

```

executed in 7ms, finished 13:58:19 2021-03-25

In [13]:
```
# Create the model

resnet_model = ResNetModel()
for x, y in train_dataset.take(1):
    y = tf.cast(y, dtype = 'int32')
    print(y.dtype)
    print(resnet_model(x))
    print(y)
    print(loss_obj(y, resnet_model(x)))
```

executed in 96ms, finished 13:59:10 2021-03-25

```
<dtype: 'int32'>
tf.Tensor(
[[0.08867583 0.08713888 0.11817738 0.10930809 0.10143802 0.08889387
  0.10437551 0.10103974 0.09937955 0.10157308]
 [0.08897601 0.09413303 0.10879799 0.10699089 0.10497347 0.09888377
  0.10212538 0.10088008 0.10562213 0.08861715]
 [0.09575275 0.10139996 0.10343292 0.1004658  0.10416432 0.09493999
  0.10095166 0.09903437 0.10337881 0.09647951]
 [0.09187554 0.09947837 0.10764831 0.10170945 0.10480637 0.09452402
  0.10315024 0.0969398  0.10637237 0.09349547]
 [0.09407722 0.09909926 0.10285106 0.10433199 0.11135953 0.09085974
  0.09960254 0.09687813 0.10591701 0.09502351]
 [0.09117531 0.0936947  0.1096489  0.10516741 0.10779335 0.09152306
  0.10156306 0.09586978 0.10501099 0.09855331]
 [0.0960987  0.09384109 0.10888263 0.11187343 0.10597844 0.08200672
  0.10606775 0.08902492 0.10329293 0.10293334]
 [0.08758024 0.0907634  0.11905275 0.10767379 0.10866065 0.08792035
  0.10654834 0.09259158 0.1064828  0.09272613]
 [0.09580077 0.09528855 0.10492529 0.09916568 0.10639544 0.09973062
  0.10018108 0.10135235 0.10112065 0.0960396 ]
 [0.09167925 0.09169737 0.11258409 0.0944254  0.11304443 0.09324434
  0.10275771 0.10609479 0.0969934  0.09747928]
 [0.08569021 0.09932481 0.10921531 0.10100949 0.10357518 0.09669838
  0.1059357  0.09956637 0.10480687 0.09417763]
 [0.08533099 0.08920892 0.12007394 0.10160306 0.10636112 0.09429563
  0.10703952 0.09626803 0.10426656 0.09555216]
 [0.09712359 0.09420418 0.11081469 0.10512327 0.10623213 0.09004286
  0.09876043 0.09561035 0.10263562 0.09945296]
 [0.09758875 0.09013408 0.10583892 0.107774   0.10510577 0.0933443
  0.09668294 0.0998717  0.10570501 0.09795458]
 [0.10347553 0.09975775 0.10184949 0.10256858 0.10222535 0.0881445
  0.10520057 0.09510516 0.10376118 0.09791186]
 [0.0902952  0.0850013  0.11982974 0.10766817 0.10371545 0.08206541
  0.10583187 0.10079671 0.10370271 0.10109339]
 [0.09452897 0.09935308 0.11404271 0.09926873 0.10382966 0.09730887
  0.10286174 0.09709357 0.10055713 0.09115554]
 [0.08335304 0.10112809 0.1068444  0.09961969 0.10801514 0.09859181
  0.10547589 0.09862088 0.10528502 0.09306609]
 [0.09358775 0.09301558 0.10909005 0.10486592 0.10462735 0.09167263
  0.10625727 0.09734409 0.10516838 0.09437101]
 [0.09808781 0.10036206 0.10600167 0.10007915 0.10253382 0.0944863
  0.10562937 0.09553531 0.10188224 0.09540237]
 [0.08885545 0.09809312 0.10604104 0.10265811 0.1071116  0.09411127
  0.10509022 0.09960042 0.10172067 0.0967181 ]
 [0.10069701 0.09983668 0.10790993 0.10210343 0.11113721 0.08955237
  0.09921454 0.09770634 0.09844545 0.09339701]
 [0.09233359 0.09488483 0.10783198 0.10786078 0.10785215 0.09603557
  0.10032237 0.09967665 0.09835221 0.0948498 ]
 [0.09357222 0.08985995 0.11201621 0.10829146 0.10912693 0.08948888
  0.10172377 0.09642685 0.10244387 0.09704987]
 [0.09465744 0.09778665 0.11321588 0.10207614 0.10483614 0.09057808
  0.10375784 0.09351192 0.10600933 0.09357064]
 [0.09219153 0.09945211 0.10727496 0.09824685 0.10654241 0.09426828
  0.10684181 0.0964018  0.10622746 0.0925528 ]
 [0.09182617 0.09696174 0.10811301 0.10494801 0.10797857 0.08910601
  0.10648773 0.09680407 0.10375878 0.09401596]
 [0.08750936 0.09363788 0.11620518 0.10758474 0.10464194 0.08902212
  0.10590972 0.09576115 0.10838769 0.09134024]
 [0.09191123 0.09916256 0.10657079 0.10082889 0.103544   0.09417554
  0.10550196 0.09927177 0.10485904 0.09417426]
 [0.09324535 0.09473272 0.11218566 0.10490228 0.10510019 0.08860381
  0.10360362 0.09702548 0.10680726 0.09379354]
 [0.10047194 0.10240545 0.09838089 0.10007896 0.10545585 0.08951013
  0.10327086 0.09526476 0.106456   0.09870519]
 [0.09511863 0.10090147 0.10170069 0.10278025 0.10618945 0.09420212
  0.10010565 0.10157481 0.09997177 0.0974552 ]], shape=(32, 10), dtype=float32)
tf.Tensor([9 0 0 3 0 2 7 2 5 5 0 9 5 5 7 9 1 0 6 4 3 1 4 8 4 3 0 2 4 4 5 3], shape=(32,), dtype=int32)
tf.Tensor(2.3153033, shape=(), dtype=float32)
```

Define the optimizer and loss function

We will use the Adam optimizer with a learning rate of 0.001, and the sparse categorical cross entropy function.

In [11]:
```
# Create the optimizer and loss

optimizer_obj = tf.keras.optimizers.Adam(learning_rate=0.001)
loss_obj = tf.keras.losses.SparseCategoricalCrossentropy()
```

executed in 4ms, finished 13:58:32 2021-03-25

Define the grad function

You should now create the `grad` function that will compute the forward and backward pass, and return the loss value and gradients that will be used in your custom training loop:

- The `grad` function takes a model instance, inputs, targets and the loss object above as arguments
- The function should use a `tf.GradientTape` context to compute the forward pass and calculate the loss
- The function should compute the gradient of the loss with respect to the model's trainable variables
- The function should return a tuple of two elements: the loss value, and a list of gradients

In [19]:
```
#### GRADED CELL ####

# Complete the following function.
# Make sure to not change the function name or arguments.

@tf.function
def grad(model, inputs, targets, loss):
    """
    This function should compute the loss and gradients of your model, corresponding to
    the inputs and targets provided. It should return the loss and gradients.
    """

```
executed in 19ms, finished 14:13:27 2021-03-25

Define the custom training loop

You should now write a custom training loop. Complete the following function, according to the spec:

- The function takes the following arguments:
  - `model` : an instance of your custom model
  - `num_epochs` : integer number of epochs to train the model
  - `dataset` : a `tf.data.Dataset` object for the training data
  - `optimizer` : an optimizer object, as created above
  - `loss` : a sparse categorical cross entropy object, as created above
  - `grad_fn` : your `grad` function above, that returns the loss and gradients for given model, inputs and targets
- Your function should train the model for the given number of epochs, using the `grad_fn` to compute gradients for each training batch, and updating the model parameters using `optimizer.apply_gradients` .
- Your function should collect the mean loss and accuracy values over the epoch, and return a tuple of two lists; the first for the list of loss values per epoch, the second for the list of accuracy values per epoch.

You may also want to print out the loss and accuracy at each epoch during the training.

In [20]:
```
#### GRADED CELL ####

# Complete the following function.
# Make sure to not change the function name or arguments.

def train_resnet(model, num_epochs, dataset, optimizer, loss, grad_fn):
    """
    This function should implement the custom training loop, as described above. It should
    return a tuple of two elements: the first element is a list of loss values per epoch, the
    second is a list of accuracy values per epoch
    """

    # Keep results for plotting
    train_loss_results, train_accuracy_results = [], []

    for epoch in range(num_epochs): # Start the training
        print("Epoch now is...", epoch)
        epoch_loss_avg = tf.keras.metrics.Mean()
        epoch_accuracy = tf.keras.metrics.CategoricalAccuracy()

        # Training loop

        for x, y in dataset:
            # Optimize the model
            loss_value, grads = grad_fn(model,x,y,loss)

            optimizer.apply_gradients(zip(grads, model.trainable_variables))

            # Compute current loss
            epoch_loss_avg(loss_value)
            # Compare predicted label to actual label
            epoch_accuracy(to_categorical(y), model(x))

        # End of epoch
        train_loss_results.append(epoch_loss_avg.result())
        train_accuracy_results.append(epoch_accuracy.result())

        print(f"Epoch: {epoch}, Loss: {epoch_loss_avg.result()}, Accuracy: {epoch_accuracy.result()}")

    return train_loss_results, train_accuracy_results
```
executed in 7ms, finished 14:13:29 2021-03-25

In [21]:
```
# Train the model for 8 epochs

train_loss_results, train_accuracy_results = train_resnet(resnet_model, 8, train_dataset, optimizer_obj, loss_obj, grad)
```
executed in 43.4s, finished 14:14:14 2021-03-25

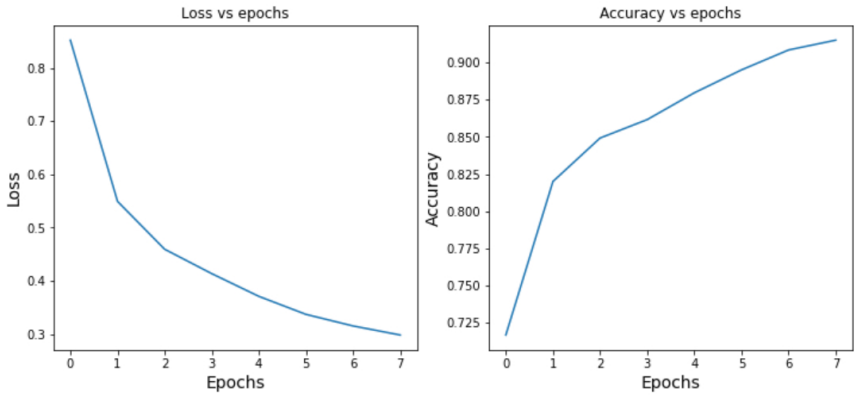
```
Epoch now is... 0
model.trainable_variables:  [<tf.Variable 'res_net_model_2/conv2d_14/kernel:0' shape=(7, 7, 1, 32) dtype=float32>, <tf.Variable
```

```
'res_net_model_2/conv2d_14/bias:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/residual_block_2/batch_normalization_8/gamma:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/residual_block_2/batch_normalization_8/beta:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/residual_block_2/conv2d_16/kernel:0' shape=(3, 3, 32, 32) dtype=float32>, <tf.Variable 'res_net_model_2/residual_block_2/conv2d_16/bias:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/residual_block_2/batch_normalization_9/gamma:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/residual_block_2/batch_normalization_9/beta:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/residual_block_2/conv2d_17/kernel:0' shape=(3, 3, 32, 32) dtype=float32>, <tf.Variable 'res_net_model_2/residual_block_2/conv2d_17/bias:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/conv2d_15/kernel:0' shape=(3, 3, 32, 32) dtype=float32>, <tf.Variable 'res_net_model_2/conv2d_15/bias:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/filters_change_residual_block_2/batch_normalization_10/gamma:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/filters_change_residual_block_2/batch_normalization_10/beta:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/filters_change_residual_block_2/conv2d_18/kernel:0' shape=(3, 3, 32, 32) dtype=float32>, <tf.Variable 'res_net_model_2/filters_change_residual_block_2/conv2d_18/bias:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/filters_change_residual_block_2/batch_normalization_11/gamma:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/filters_change_residual_block_2/batch_normalization_11/beta:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/filters_change_residual_block_2/conv2d_19/kernel:0' shape=(3, 3, 32, 64) dtype=float32>, <tf.Variable 'res_net_model_2/filters_change_residual_block_2/conv2d_19/bias:0' shape=(64,) dtype=float32>, <tf.Variable 'res_net_model_2/filters_change_residual_block_2/conv2d_20/kernel:0' shape=(1, 1, 32, 64) dtype=float32>, <tf.Variable 'res_net_model_2/filters_change_residual_block_2/conv2d_20/bias:0' shape=(64,) dtype=float32>, <tf.Variable 'res_net_model_2/dense_2/kernel:0' shape=(1600, 10) dtype=float32>, <tf.Variable 'res_net_model_2/dense_2/bias:0' shape=(10,) dtype=float32>]
model.trainable_variables:  [<tf.Variable 'res_net_model_2/conv2d_14/kernel:0' shape=(7, 7, 1, 32) dtype=float32>, <tf.Variable 'res_net_model_2/conv2d_14/bias:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/residual_block_2/batch_normalization_8/gamma:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/residual_block_2/batch_normalization_8/beta:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/residual_block_2/conv2d_16/kernel:0' shape=(3, 3, 32, 32) dtype=float32>, <tf.Variable 'res_net_model_2/residual_block_2/conv2d_16/bias:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/residual_block_2/batch_normalization_9/gamma:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/residual_block_2/batch_normalization_9/beta:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/residual_block_2/conv2d_17/kernel:0' shape=(3, 3, 32, 32) dtype=float32>, <tf.Variable 'res_net_model_2/residual_block_2/conv2d_17/bias:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/conv2d_15/kernel:0' shape=(3, 3, 32, 32) dtype=float32>, <tf.Variable 'res_net_model_2/conv2d_15/bias:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/filters_change_residual_block_2/batch_normalization_10/gamma:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/filters_change_residual_block_2/batch_normalization_10/beta:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/filters_change_residual_block_2/conv2d_18/kernel:0' shape=(3, 3, 32, 32) dtype=float32>, <tf.Variable 'res_net_model_2/filters_change_residual_block_2/conv2d_18/bias:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/filters_change_residual_block_2/batch_normalization_11/gamma:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/filters_change_residual_block_2/batch_normalization_11/beta:0' shape=(32,) dtype=float32>, <tf.Variable 'res_net_model_2/filters_change_residual_block_2/conv2d_19/kernel:0' shape=(3, 3, 32, 64) dtype=float32>, <tf.Variable 'res_net_model_2/filters_change_residual_block_2/conv2d_19/bias:0' shape=(64,) dtype=float32>, <tf.Variable 'res_net_model_2/filters_change_residual_block_2/conv2d_20/kernel:0' shape=(1, 1, 32, 64) dtype=float32>, <tf.Variable 'res_net_model_2/filters_change_residual_block_2/conv2d_20/bias:0' shape=(64,) dtype=float32>, <tf.Variable 'res_net_model_2/dense_2/kernel:0' shape=(1600, 10) dtype=float32>, <tf.Variable 'res_net_model_2/dense_2/bias:0' shape=(10,) dtype=float32>]
Epoch: 0, Loss: 0.2656477987766266, Accuracy: 0.9222000241279602
Epoch now is... 1
Epoch: 1, Loss: 0.23948127031326294, Accuracy: 0.9355999827384949
Epoch now is... 2
Epoch: 2, Loss: 0.2324903905391693, Accuracy: 0.9430000185966492
Epoch now is... 3
Epoch: 3, Loss: 0.21865154802799225, Accuracy: 0.9472000002861023
Epoch now is... 4
Epoch: 4, Loss: 0.19873179495334625, Accuracy: 0.9545999765396118
Epoch now is... 5
Epoch: 5, Loss: 0.17139625549316406, Accuracy: 0.9607999920845032
Epoch now is... 6
Epoch: 6, Loss: 0.1709071546792984, Accuracy: 0.9649999737739563
Epoch now is... 7
Epoch: 7, Loss: 0.16152715682983398, Accuracy: 0.9661999940872192
```

Plot the learning curves

In [17]:
```
fig, axes = plt.subplots(1, 2, sharex=True, figsize=(12, 5))

axes[0].set_xlabel("Epochs", fontsize=14)
axes[0].set_ylabel("Loss", fontsize=14)
axes[0].set_title('Loss vs epochs')
axes[0].plot(train_loss_results)

axes[1].set_title('Accuracy vs epochs')
axes[1].set_ylabel("Accuracy", fontsize=14)
axes[1].set_xlabel("Epochs", fontsize=14)
axes[1].plot(train_accuracy_results)
plt.show()
```

executed in 296ms, finished 14:09:02 2021-03-25



Evaluate the model performance on the test dataset

In [18]:
```
# Compute the test loss and accuracy

epoch_loss_avg = tf.keras.metrics.Mean()
epoch_accuracy = tf.keras.metrics.CategoricalAccuracy()
```

```
5
6  for x, y in test_dataset:
7      model_output = resnet_model(x)
8      epoch_loss_avg(loss_obj(y, model_output))
9      epoch_accuracy(to_categorical(y), model_output)
10
11 print("Test loss: {:.3f}".format(epoch_loss_avg.result().numpy()))
12 print("Test accuracy: {:.3%}".format(epoch_accuracy.result().numpy()))
```

executed in 3.75s, finished 14:09:12 2021-03-25

```
Test loss: 0.530
Test accuracy: 83.110%
```

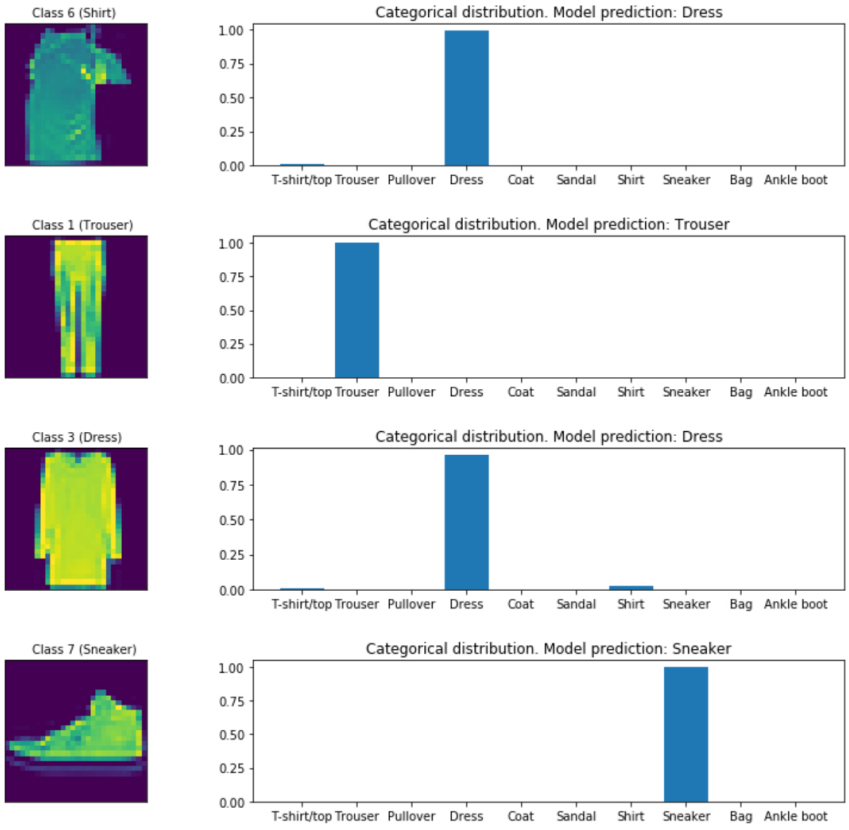
Model predictions

Let's see some model predictions! We will randomly select four images from the test data, and display the image and label for each.

For each test image, model's prediction (the label with maximum probability) is shown, together with a plot showing the model's categorical distribution.

In [65]:

```
1  # Run this cell to get model predictions on randomly selected test images
2
3  num_test_images = test_images.shape[0]
4
5  random_inx = np.random.choice(test_images.shape[0], 4)
6  random_test_images = test_images[random_inx, ...]
7  random_test_labels = test_labels[random_inx, ...]
8
9  predictions = resnet_model(random_test_images)
10
11 fig, axes = plt.subplots(4, 2, figsize=(16, 12))
12 fig.subplots_adjust(hspace=0.5, wspace=-0.2)
13
14 for i, (prediction, image, label) in enumerate(zip(predictions, random_test_images, random_test_labels)):
15     axes[i, 0].imshow(np.squeeze(image))
16     axes[i, 0].get_xaxis().set_visible(False)
17     axes[i, 0].get_yaxis().set_visible(False)
18     axes[i, 0].text(5., -2., f'Class {label} ({image_labels[label]})')
19     axes[i, 1].bar(np.arange(len(prediction)), prediction)
20     axes[i, 1].set_xticks(np.arange(len(prediction)))
21     axes[i, 1].set_xticklabels(image_labels, rotation=0)
22     pred_inx = np.argmax(prediction)
23     axes[i, 1].set_title(f"Categorical distribution. Model prediction: {image_labels[pred_inx]}")
24
25 plt.show()
```



Congratulations for completing this programming assignment! You're now ready to move on to the capstone project for this course.