CONQUERING FASHION MNIST

Using CNNs and Computer Vision

ISHAN PARASHAR

Problem Statement Analysis: Fashion MNIST

- Fashion-MNIST is a dataset of Zalando's article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes.
- Since this dataset is already predefined in terms for test and train split, it is not for the end user to decide the training and testing dataset.
- Use of Convolutional Neural Networks is extremely important and their application in Computer Vision is ubiquitous.
- Use of Intel’s various optimizations in the AI Analytics toolkit will be useful in the model training and evaluating process.

Structure of The Model:

- The following is an image created using Visualkeras library in Python to help visualize the structure of the model.
- The next slide contains the details of various layers used in creating this CNN based model as well as the total number of parameters.
- The best target accuracy of the model trained after various complete iterations on different environments and time periods has been saved.
- For optimizer, I have used **Adam**, and for loss **categorical cross entropy**.

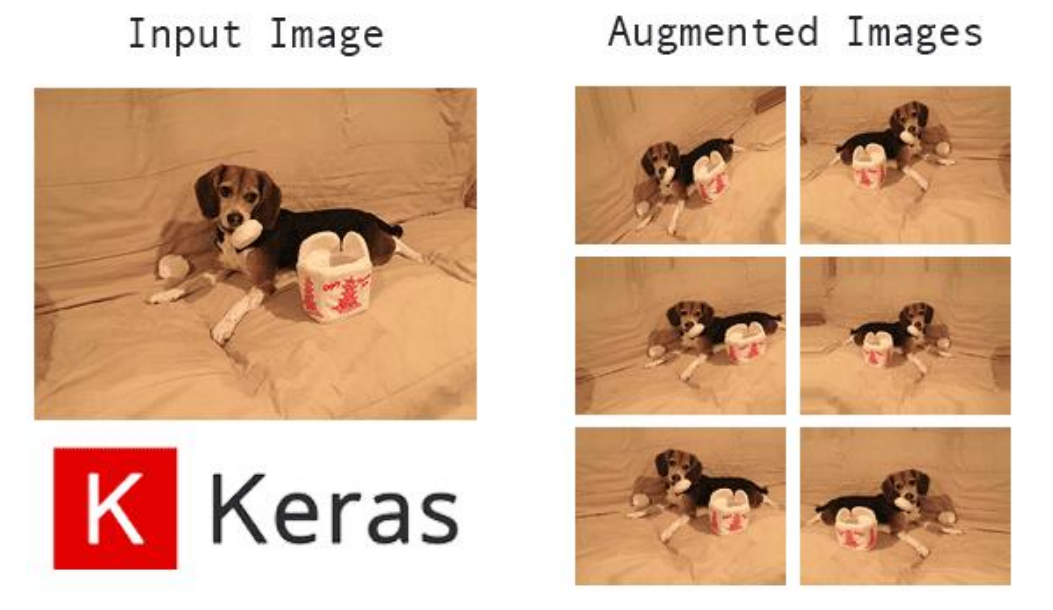
Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 64)	640
conv2d_1 (Conv2D)	(None, 28, 28, 64)	36928
module_wrapper (ModuleWrappe	(None, 28, 28, 64)	256
max_pooling2d (MaxPooling2D)	(None, 14, 14, 64)	0
dropout (Dropout)	(None, 14, 14, 64)	0
conv2d_2 (Conv2D)	(None, 14, 14, 128)	73856
conv2d_3 (Conv2D)	(None, 14, 14, 128)	147584
module_wrapper_1 (ModuleWrap	(None, 14, 14, 128)	512
max_pooling2d_1 (MaxPooling2	(None, 7, 7, 128)	0
dropout_1 (Dropout)	(None, 7, 7, 128)	0
conv2d_4 (Conv2D)	(None, 7, 7, 256)	295168
conv2d_5 (Conv2D)	(None, 7, 7, 256)	590080
conv2d_6 (Conv2D)	(None, 7, 7, 256)	590080
module_wrapper_2 (ModuleWrap	(None, 7, 7, 256)	1024
max_pooling2d_2 (MaxPooling2	(None, 3, 3, 256)	0
dropout_2 (Dropout)	(None, 3, 3, 256)	0
flatten (Flatten)	(None, 2304)	0
dense (Dense)	(None, 2048)	4720640
dropout_3 (Dropout)	(None, 2048)	0
dense_1 (Dense)	(None, 512)	1049088
dense_2 (Dense)	(None, 10)	5130

Total params: 7,510,986
Trainable params: 7,510,090
Non-trainable params: 896

Structure of the Model:

- The **training code** is available as a Jupyter notebook titled **‘training_code.ipynb’** in the **code** folder. Running the training just once or twice might not give you the same test accuracy as the saved model but very close to it **(94.5 to 94.8 %range)**
- As you can see the total no. of parameters is approximately **7.5 million**

Data Preprocessing and Augmentation Process:



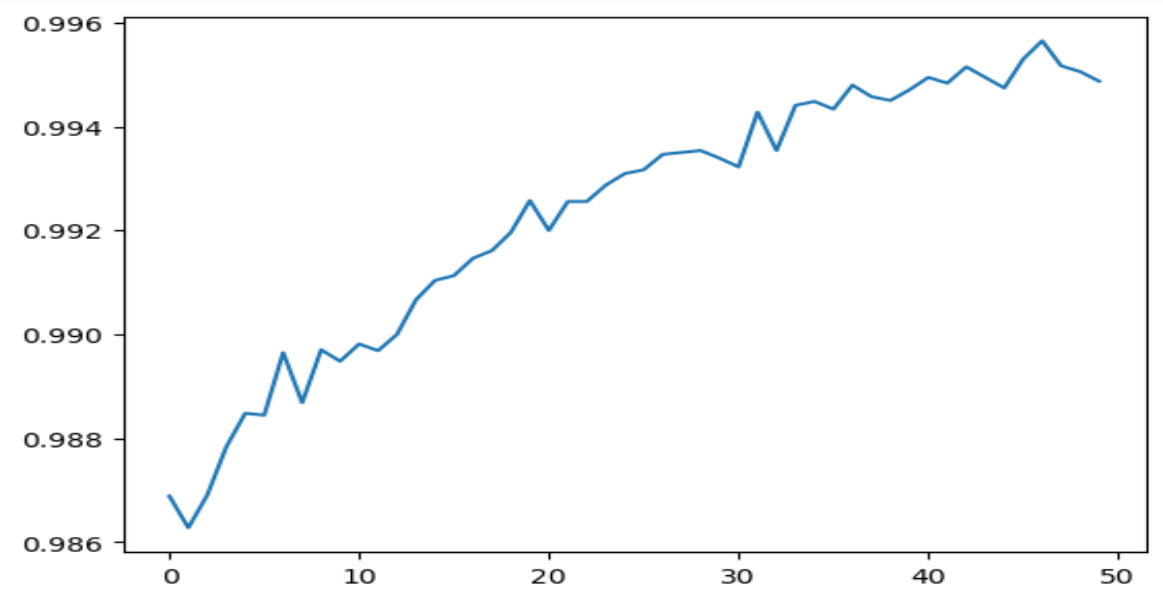
- The data preprocessing is rather simple as it is a relatively uncomplicated dataset with no nan values. I have used **normalization of data**, i.e., divided the images array values by 255 and **one hot encoded** the labels.
- The **data augmentation** plays a key role in achieving a **higher desired accuracy.**
- For data augmentation I have used Keras Image Data Generator to create tensors for augmentation of the training data set.
- The **ImageDataGenerator** is a “in-place” and “on-the-fly” data augmentation because this augmentation is done at training time (i.e., we are not generating these examples ahead of time/prior to training).

```
datagen = tf.keras.preprocessing.image.ImageDataGenerator(
    featurewise_center=True,
    featurewise_std_normalization=True,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True)
```

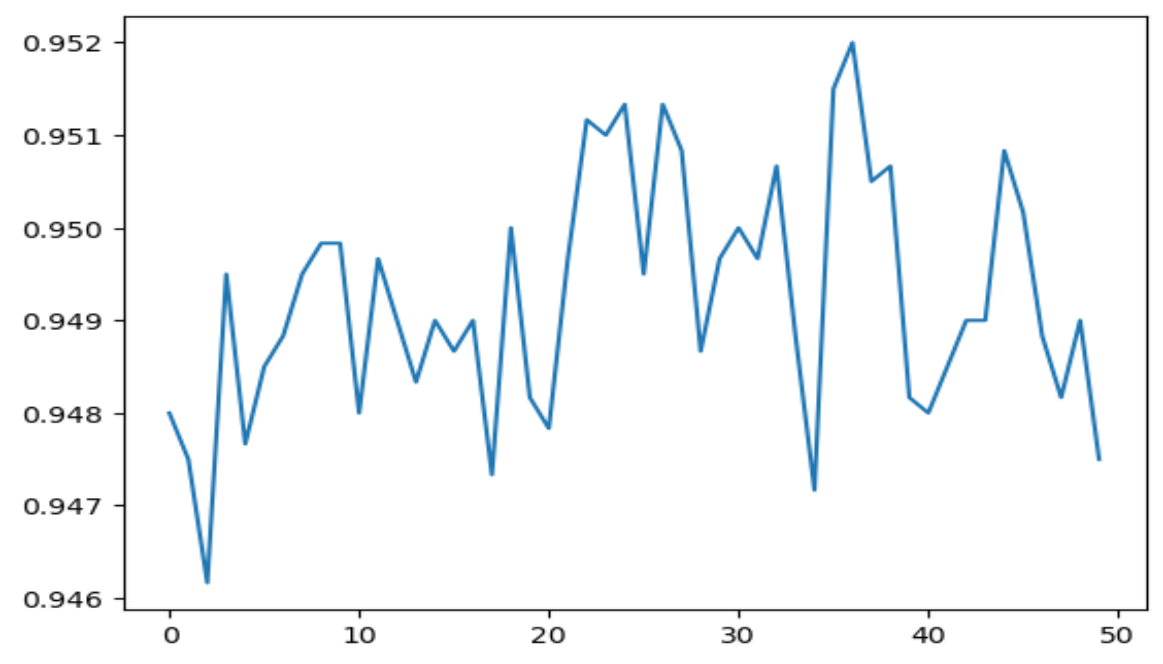
```
datagen.fit(X_train)
```

Model Training Process:

- In the training process in-addition to the usual ‘model.fit’ I have incorporated **Early Stopping** and **Reduce Learning Rate on Plateau**.
- These provide a significant boost to accuracy during the training process.
- The batch size is chosen as 32 and number of epochs trained for is 50



Graph of training accuracy vs epochs



Graph of validation accuracy vs epochs

Intel oneAPI optimization : Model Training

Intel's optimization for Tensorflow has one key feature i.e Graph Optimization, and to test this I have used Intel's DevCloud environment and the results have been amazing.

For comparison I have used a similar environment i.e. a cloud based environment which doesn't have Intel's optimization for Tensorflow and is running a vanilla build of Tensorflow.

The first image is of DevCloud and second of Google Colab, the difference in per epoch time is visible. Intel's Tensorflow is atleast 8.5x faster than vanilla Tensorflow

```
2023-07-14 05:40:54.546782: I tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:114] Plugin optimizer for device_type CPU is enabled.
2023-07-14 05:40:54.555434: I tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:114] Plugin optimizer for device_type CPU is enabled.
2023-07-14 05:40:54.569220: I tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:114] Plugin optimizer for device_type CPU is enabled.
1688/1688 [==============================] - 111s 66ms/step - loss: 0.0382 - accuracy: 0.9863 - val_loss: 0.2748 - val_accuracy: 0.9447
2023-07-14 05:42:42.472237: I tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:114] Plugin optimizer for device_type CPU is enabled.
2023-07-14 05:42:42.480675: I tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:114] Plugin optimizer for device_type CPU is enabled.
2023-07-14 05:42:42.494227: I tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:114] Plugin optimizer for device_type CPU is enabled.
1688/1688 [==============================] - 108s 64ms/step - loss: 0.0398 - accuracy: 0.9855 - val_loss: 0.2841 - val_accuracy: 0.9445
2023-07-14 05:44:32.877134: I tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:114] Plugin optimizer for device_type CPU is enabled.
2023-07-14 05:44:32.885704: I tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:114] Plugin optimizer for device_type CPU is enabled.
2023-07-14 05:44:32.899196: I tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:114] Plugin optimizer for device_type CPU is enabled.
1688/1688 [==============================] - 110s 65ms/step - loss: 0.0379 - accuracy: 0.9867 - val_loss: 0.2818 - val_accuracy: 0.9440
```

```
Epoch 4/10
1687/1687 [==============================] - 941s 557ms/step - loss: 0.4559 - accuracy: 0.8382
Epoch 5/10
1687/1687 [==============================] - 924s 548ms/step - loss: 0.4236 - accuracy: 0.8478
Epoch 6/10
1687/1687 [==============================] - 917s 543ms/step - loss: 0.4012 - accuracy: 0.8573
Epoch 7/10
1687/1687 [==============================] - 930s 551ms/step - loss: 0.3779 - accuracy: 0.8675
Epoch 8/10
```

Final Results:

- The best test accuracy of the model after various complete training iterations on different environments and time periods is approximately 95**%.**
- The testing code to access the saved model and evaluate it is available as a Jupyter notebook titled **‘testing_demo.ipynb’** in the **code** folder.
- Intel’s optimizations provide upto 9x boost on model training speeds compared to vanilla builds of Tensorflow.

References:

1. Keras ImageDataGenerator and Data Augmentation – PyImageSearch

2. A Practical Introduction to Keras Callbacks in TensorFlow 2 | by B. Chen | Towards Data Science

3. 8 Simple Techniques to Prevent Overfitting | by David Chuan-En Lin | Towards Data Science

4. Convolutional Neural Network (CNN) | TensorFlow Core