Jakub Czerny
Fashion MNIST

DEMO:

[Link to demo]

Final results:

94.22% on average over 5 repeated trainings

94.86% ensemble voting of 5 models

Repo:

[fashion-mnist-assignment]

`cd` to project home directory `/fashion-mnist-assignment`

run `jupyter-notebook` from there (to make sure the paths & directories work as expected)

you can find main jupyter notebook in in `/my_code/training.ipynb` together with python scripts

---

Environment

Python3

All the experiments were run on Google Colab using GPU

Requirements: `pip install -r requirements.txt`

Run demo:

`cd` to `/fashion-mnist-assignment/my_code`
Run `python camera.py`

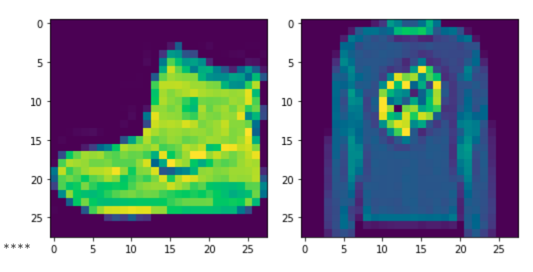
Data processing & augmentation

As per usual, I started by exploring the data and checking the classes distribution. This involved understanding the data (I noticed that there's little padding around the pictures, and the classification ss rather difficult task even for people - only 83.5% accuracy).
Then, I moved to data preparation - reshaping, split, scaling, adding channel dimension, one-hot encoding labels.

At first, the preprocessing was just data rescaling to range 0-1. It's a reasonable choice for the images, being bounded data. I ended up adding only following 3 transformations:

- Width shift (3px)
- Height shift (3px)
- Horizontal flip

Although simple, they helped regularize the training (shifts were very small - but since sometimes there was no padding around the objects, the characteristic regions (like the colar on the right picture) could end up being out of the picture. (forced the network to explore more features of objects)



****

I also tried seemingly reasonable transformations such as:

- Rotation

- Shear

- Zooming

but all of them led to worse performance. I guess rotation and shearing distorted already pixelated image too much and zooming would cut out too much - could happen that important features from both sides or top/bottom would be lost. I tried these ideas on v1 model - and they would make it difficult to even overfit the data - which might indicate that transformation is not proper for the data.

Effects of data augmentation:

model	layers	params	augment	Acc. train	Acc. test	Acc. val	Note
v1	8	260k	False	93.9	90.6	91.4	Early stopped
v1	8	260k	False	96.7	91.3	91.7	30 epochs
v1	8	260k	True	92.5	92.0	92.4	30 epochs

Regardless of whether overfit or early stopped, the model without augmentation performs worse when testing.

Models

more in `my_code/models.py`

My first thought was transfer learning, but since the images were grayscale, it would require some extra steps, such as duplicating the intensity into 3 channels to mimic RGB. Rather dummy idea - more computationally expensive and probably not worth the effort.

The first model I typically try for images is VGG-like architecture, so I built a simplified version of it `model v1`.
Then I manually iterated multiple times increasing the capacity and regularizing it.

Built models:

- VGG-like (`v1` , `v2` , `v3` , `v4`)

All 4 models have pretty much the same backbone - the deeper into the network the more filter the convolutional layers have i.e.
64 - 64 - 128 - 128 -256 -256

Key difference: number of dense layers (2 or 3), dropouts, batch normalization, ReLU / LeakyReLU

Achieved accuracy of ~93.5%

- Own model (`v5`)

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_30 (Conv2D)           (None, 28, 28, 64)        640
batch_normalization_30 (Batc (None, 28, 28, 64)        256
conv2d_31 (Conv2D)           (None, 28, 28, 128)       73856
max_pooling2d_15 (MaxPooling (None, 14, 14, 128)       0
dropout_25 (Dropout)         (None, 14, 14, 128)       0
batch_normalization_31 (Batc (None, 14, 14, 128)       512
conv2d_32 (Conv2D)           (None, 14, 14, 64)        73792
batch_normalization_32 (Batc (None, 14, 14, 64)        256
conv2d_33 (Conv2D)           (None, 14, 14, 128)       73856
max_pooling2d_16 (MaxPooling (None, 7, 7, 128)         0
dropout_26 (Dropout)         (None, 7, 7, 128)         0
batch_normalization_33 (Batc (None, 7, 7, 128)         512
conv2d_34 (Conv2D)           (None, 7, 7, 64)          73792
batch_normalization_34 (Batc (None, 7, 7, 64)          256
conv2d_35 (Conv2D)           (None, 5, 5, 128)         73856
max_pooling2d_17 (MaxPooling (None, 2, 2, 128)         0
dropout_27 (Dropout)         (None, 2, 2, 128)         0
flatten_5 (Flatten)          (None, 512)               0
batch_normalization_35 (Batc (None, 512)               2048
dense_15 (Dense)             (None, 512)               262656
dropout_28 (Dropout)         (None, 512)               0
dense_16 (Dense)             (None, 512)               262656
dropout_29 (Dropout)         (None, 512)               0
dense_17 (Dense)             (None, 10)                5130
=================================================================
Total params: 904,074
Trainable params: 902,154
Non-trainable params: 1,920
```

Specification:

- 905k parameters - I was trying to keep it lean

- 6 convolutional & 3 dense layers

- Average of 94.22% accuracy over 5 repetitions (ensemble voting 94.86%)

- Inference speed: ~100 images/per second using Google Colab CPU (sequentially feeding images one by one) ([Google colab spec])

- ~10Mb model

Architecture:

It's been discussed quite a bit in the papers that 3x3 convolutions are usually enough (even for more complicated tasks e.g. semantic seg), while keeping the number of parameters low - so I only built models using 3x3 kernels.

The network is built up of 3 "convolutional modules" and 3 dense layers.

Each convolutional module has following units:
`Conv3x3 (64) - BatchNorm - Conv3x3(128) - MaxPool2x2 - Dropout(0.2) - BatchNorm`

They are followed by:
`Dense (512) - Dropout(0.5) - Dense (512) - Dropout(0.5) - Dense (10)`

Then, all dense layers are further regularized with L2 reg. applied to their weights (very small lambda 1e-5), but should stop weights from getting very big, what in turn improves generalization and prevents overfitting.

Notes:

I got the results by training the models for 100 epochs and saving the once with best validation losses. I repeated that 5 times and calculated average performance on test dataset. I could probably get a bit higher accuracy with hyperparameter optimization (which I implemented the pipeline for - Bayesian optimization), but at some point Google Colab took away my GPU as I was using it too much.

Even though, the overfitting began already after 30-40 epochs, after trying early stopping, it turned out it was beneficial to keep going and strongly overfit the data.

```
Epoch 00036: val_loss did not improve from 0.18893
97/97 - 21s - loss: 0.1950 - acc: 0.9338 - val_loss: 0.2013 - val_acc: 0.9351
Epoch 37/100
Epoch 1/100

Epoch 00037: val_loss did not improve from 0.18893
97/97 - 21s - loss: 0.1933 - acc: 0.9341 - val_loss: 0.1895 - val_acc: 0.9382
Epoch 38/100
Epoch 1/100

Epoch 00038: val_loss did not improve from 0.18893
97/97 - 21s - loss: 0.1878 - acc: 0.9366 - val_loss: 0.1947 - val_acc: 0.9362
Epoch 39/100
Epoch 1/100

Epoch 00039: val_loss did not improve from 0.18893
97/97 - 21s - loss: 0.1902 - acc: 0.9351 - val_loss: 0.1901 - val_acc: 0.9380
Epoch 40/100
Epoch 1/100

Epoch 00040: val_loss did not improve from 0.18893
97/97 - 21s - loss: 0.1833 - acc: 0.9375 - val_loss: 0.1974 - val_acc: 0.9346
Epoch 41/100
Epoch 1/100
```

The losses and metrics around 100th epoch. This was not only better on validation but also test dataset. When doing early stopping 5 epochs after the loss hasn't gone down I would always end up with accuracy in the proximity of 93.5%.

```
Epoch 00097: val_loss did not improve from 0.16918
97/97 - 20s - loss: 0.1137 - acc: 0.9622 - val_loss: 0.1783 - val_acc: 0.9457
Epoch 98/100
Epoch 1/100

Epoch 00098: val_loss did not improve from 0.16918
97/97 - 20s - loss: 0.1119 - acc: 0.9626 - val_loss: 0.1774 - val_acc: 0.9479
Epoch 99/100
Epoch 1/100

Epoch 00099: val_loss did not improve from 0.16918
97/97 - 20s - loss: 0.1132 - acc: 0.9619 - val_loss: 0.1877 - val_acc: 0.9465
Epoch 100/100
Epoch 1/100

Epoch 00100: val_loss did not improve from 0.16918
97/97 - 21s - loss: 0.1103 - acc: 0.9626 - val_loss: 0.1855 - val_acc: 0.9443
```

Evaluation of the trained models `v5`:

```
X_test, y_test = mnist_reader.load_mnist('data/fashion', kind='t10k')
X_test = X_test.reshape(X_test.shape[0], IMG_SIZE, IMG_SIZE, 1)
y_test_encoded = to_categorical(y_test, num_classes=NUM_CLASSES, dtype='float32')

test_generator_args = dict(
    data_format = 'channels_last',
    rescale=1./255,
)

test_datagen = ImageDataGenerator(**test_generator_args)
test_datagen.fit(X_test)

test_generator = test_datagen.flow(
    X_test,
    y_test_encoded,
    batch_size=250,
    shuffle=False
)

model_name = 'v5'

accuracies = []
for i in range(5):
    model = tf.keras.models.load_model('models/model_{:}_{:}.h5'.format(model_name,i))

    result = model.evaluate_generator(test_generator)
    accuracies.append(result[1])
    print("Model {:} accuracy: {:.4f}".format(i,result[1]))

print("\nMean accuracy {:.4f}".format(np.mean(accuracies)))
```

```
Model 0 accuracy: 0.9398
Model 1 accuracy: 0.9419
Model 2 accuracy: 0.9431
Model 3 accuracy: 0.9407
Model 4 accuracy: 0.9457

Mean accuracy 0.9422
```

Since I already had 5 models, I decided to ensemble them by simple voting scheme. I am aware that this kind of strategy works best in case the models have different structures / are of different kinds so that they have different misclassification distribution - making it actually possible to catch them. Nonetheless, there's been improvement of over 0.6%, and their combination turned out better than any of the models alone. Cool stuff.

```
Test accuracy 0.9486

[[911   1  18   8   0   1  56   0   5   0]
 [  0 994   0   5   0   0   0   0   1   0]
 [ 17   1 926   8  20   0  28   0   0   0]
 [  9   2   6 952  15   0  15   0   1   0]
 [  0   1  16  10 941   0  32   0   0   0]
 [  0   0   0   0   0 989   0   8   0   3]
 [ 81   0  33  14  41   0 829   0   2   0]
 [  0   0   0   0   0   3   0 985   0  12]
 [  0   0   0   2   0   0   0   0 998   0]
 [  0   0   0   0   0   6   0  33   0 961]]
```

Future idea:

It could be interesting to collapse t-shirt & pullover & shirt (classes 0,2,6) into one and consider it as 8-class problem, and then build a separate model for discrimination of the three.

Other models I tried:

- bottleneck ResNet (like ResNet50) (`model v6`)

Model is built of bottleneck modules:
1x1 conv - keep spatial size but change number of filters, commonly smaller than input
3x3 conv - classical convolution operation, commonly the same number of filters as above
1x1 conv - remap the data into original size - number of channels - allows easy residual connection

Idea behind this structure is to force the network to find more compact representation of the data - thus making it focus on really meaningful properties that carry same amount of information.

- simple ResNet (like ResNet18) (`model v7`)

Reminds VGG architecture but with residual connections

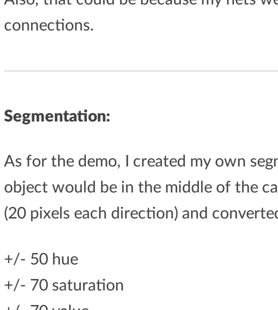
Idea from [ResNet] paper
[Blog post] on ResNets

I didn't notice improvement, but then I haven't tried overfitting them as the other ones. They were more computationally demanding.
Also, that could be because my nets were rather shallow architectures and didn't benefit much from skip connections.

Segmentation:

As for the demo, I created my own segmentation algorithm using mostly openCV. Basic idea was that the object would be in the middle of the camera view. I then took an average "color" around the central pixel (20 pixels each direction) and converted it to HSV. Then I found the rest of the object within range

+/- 50 hue
+/- 70 saturation
+/- 70 value



Then I applied closing morphological operation (dilation followed by erosion 5 times) to fill up the gaps and make the segmentation smoother. Next, I used openCV to detect contours and selected the one with the biggest area containing the central pixel.

Then I extracted the bounding box given the contour, cropped out the object and downscaled it to 28x28.

The major drawback of this approach is the fact that it expects somewhat uni-color objects.