DEMO:
https://drive.google.com/file/d/1_4bvglwhTv0_P-BhrG4WdfYNN8W-1ZaX/view?usp=sharing

Final results:
94.22% on average over 5 repeated trainings
94.86% ensemble voting of 5 models

Repo:
https://github.com/JakubCzerny/fashion-mnist-assignment/blob/master/README.md
`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

Developed in Python3
All requirements: `pip install -r requirements.txt`

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

Data processing & augmentation

As per usual, I started by exploring the data and checking the class distribution. Then, I moved to data preparation (data split, adding channel dimension, one-hot encoding labels and pre-processing)

At first, the preprocessing was just data rescaling to range 0-1. Then I tested basic transformations such as:

- Width shift
- Height shift
- Horizontal flip

which helped regularize the training (shifts were very small - but since sometimes there was no padding around the objects, the characteristic regions 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 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 performs better when trained with data augmentation.

Models
(check my_code/models.py)

My first thought was transfer learning, but since the images were grayscale, it would need some extra steps, such as duplicating the intensity 3 times, to make it 3-channel. Dummy idea - more computationally expensive and probably not worth the effort.

The first model I typically try for images is VGG, so I built a simplified version of it (model v1).
Then I iterate multiple times increasing the capacity and regularizing it.
Built models:

- VGG like (v1, v2, v3, v4)
  All 4 models have pretty much the same backbones - the deeper into the network the more filter the convolutional layers have
  Key difference: number of dense layers, 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
- 9 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) Google Colab spec (https://colab.research.google.com/drive/151805XTDg--dgHb3-AXJCpnWaqRhop_2)
- ~10Mb model

Model:

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 has 3 "convolutional modules" and 3 dense layers. The convolutional modules are:

Conv3x3 (64)
BatchNorm
Conv3x3(128)
MaxPool2x2
Dropout(0.2)

Then in between the dense layers I added a dropout of 0.5, and further regularized the net with L2 reg. applied to the kernels of dense layers (very small lambda 1e-5) - this should stop weights from getting very big, what in turn should help with generalization.

Notes:
I got the results by training the models for 100 epochs and saving a model with best validation loss (using validation dataset). 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), but google colab took away my GPU as I was using it too much.

```
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.

```
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]]
```

I also tried 2 residual models:

- bottleneck ResNet (like ResNet50) (model v6)

  Model is built of bottleneck modules:
  1) 1x1 conv - keep spatial size but change number of filters, commonly smaller than input
  2) 3x3 conv - classical convolution operation, commonly the same number of filters as above
  3) 1x1 conv - remap the data into original size - number of channels - allows easy residual connection

- simple ResNet (like ResNet18) (model v7)
  Remind VGG architecture but with residual connections

Idea from paper
https://arxiv.org/pdf/1512.03385.pdf

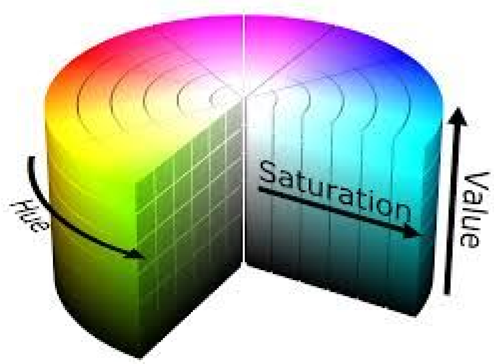
Blog post:
https://medium.com/@14prakash/understanding-and-implementing-architectures-of-resnet-and-resnext-for-state-of-the-art-image-cc5d0adf648e

I didn't notice improvement. 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 way) and converted to HSV. Then I found the rest of the object within range

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



Then I applied closing morphological operation to fill up the gaps. Next, I used openCV to detect contours and selected 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.

Jakub Czerny