Table of Content:

- 1. Introduction
- 2. Data Preprocessing
  - 2.1. Data Augmentation
  - 2.2. Train, Dev and Test sets
- 3. Models
  - 3.1. Metrics and Optimization
- 4. Experiments
  - 4.1. Comparing Architectures
  - 4.2. Comparing Loss Functions
  - 4.3. Data Augmentation Impact
- 5. Final Model
- 6. Demo Program
- 7. Summary

1. Introduction

In this work we're trying to build a classifier for *Fashion mnist* dataset. We carried out few experiments to get a notion in which direction to go in order to find as good as possible classifier. Though, we haven't tried every idea we come up with because of the time limit we will describe some of them, present corresponding results and propose ideas for further experiments and impovements.

2. Data Preprocessing

Fashion mnist dataset is already cleaned and uniformly balanced (all classes are equaly present in both training and test parts) so no additional preprocessing has been carried out. From obvious technical reasons we were unable to check corectness of provided labels though for benchmark datasets like this one chances for mislabeled examples are fairly low.

In the rest of this section we are going to describe our augmentation methods and data splitting procedure for training and evaluation of our models.

2.1. Data Augmentation

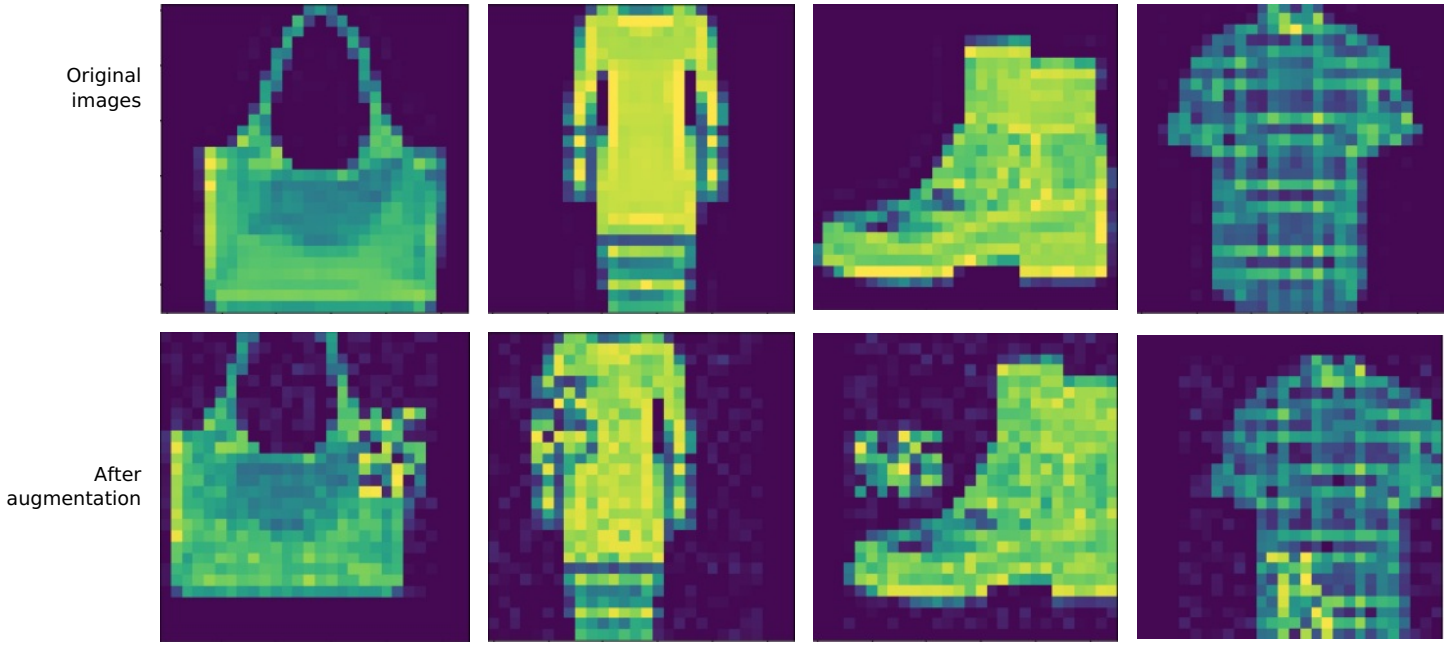
Augmentation methods we used are (in brackets are given functions that implement corresponding augmentation method in *data_input.py* sctipt):

- Random noise (add_noise). Adds random gausian noise to all pixels in the input image. Note: since all pixels that don't belong to the object have value of zero, it is worth investigating adding random noise just to non-zero pixels. However, this effect hasn't been tested.
- Adjust brightness (adjust_brightness). Values of all pixels in the image are either increased or decreased by a random percentage obtained from fixed predifined range.
- Horizontal flip (hotizontal_flip). Randomly filps image in the horizontal axis. Sandals, sneakers and ankle boots are not flipped since they are oriented in the same direction in both training and test sets. Flipping those classes will undoubtly make model more robust in general but it won't perform necessarily better on test set.
- Random patch erasing (erase_patch). This augmentatino method has been described in the following publication: Random Erasing Data Augmentation (https://arxiv.org/pdf/1708.04896v2.pdf). It assumes randomly erasing (masking) rectangles in the input image. Size of rectangles as well as its aspect ratio is obtained randomly from predefined ranges. For each image different values are chosen.
- Random cropping (crop). The input resolution is only 28x28 pixels so standard random cropping procedure won't be suitable for this problem. Instead we used random cropping approach also described in the previously mentioned paper. This method adds padding around the input image and than from the broadened image randomly crops image of the same resolution as the original image. In our case we add padding of 4 pixels (ending up with an image of size 36x36) and then we perform random cropping to get image of size 28x28 pixels.

All augmentation methods are applied online to each image independetly. Probability of applying each method is given by AUGMENT_PROB parameter (see *config.json* file). For example, if AUGMENT_PROB is equal to p, then probability of a given image remaining unchanged is (1-p)^5 because we have 5 augmentation options.

Here are some examples after applying all augmentation methods except hortizontal flip:

-	Bag (class 8)	Dress (class 3)	Ankle boot (class 9)	Shirt (class 6)



2.2. Train, Dev and Test sets

Fashion mnist dataset consits of two parts: train part - 60000 images and test part - 10000 images. In both parts classes are equaly distributed (6000 examples of each class in train part and 1000 images of each class in test part). Since objective of this work is to evaluate performance of the model on test part, it MUST NOT be included in training procedure by any means. Thus, we separate fashion mnist train part into 2 parts: (ours) train part (used for optimizing model parameters) and validation or development part (used for model evaluation during training so we could addopt early stopping strategy). Validation set is obtained from train part (fashion mnist train set) by stratified sampling of 4000 examples (uniform class distribution is preserved).

3. Models

For the warm-up we employed 2 rather simplistic architectures consisting of stacked convolutional and pooling layers (they resembles old-fashioned architectures like AlexNet or VGG). Since we deal with very low input resolution, we opted for rather big filter size in simple_model_2 to test if we can directly learn mid or high-level features. However, without proper kernel visualisation we are unable to confirm this claim eventhough simple_model_2 significantly outperformed simple_model_1 (which might be due to bigger number of parameters).

We also tested several well-known modern architectures: ResNet, Inception and Inception-ResNet. However, any version of these modern architectures are rather big for the problem we are solving here: size of our images is just 28x28x1 and we have 10 classes only. What confirms this statement is the fact that one of our simplistic models (described bellow) consisting of just few 3x3 convolutions achieves quite descent results on validation set (accuracy goes over 92%). Thus, in order to get appropriate architecture for this task we reduced the size of original network by using shalower networks with less channels, but we keep building blocks of these networks unchanged.

Architecture of all models is given bellow. Before that, we list several features common for all models:

- All models apply batch normalization.
- All models end with 2 fully connected layer. Middle layer has 128 nodes.
- Relu activation function is used for all layers except for the last one.
- Batch size is 64.

Now, we will describe each of the models in more detail:

- **simple_model_1** (114k parameters)
  - 3x3 convolution; stride 1; valid padding; 8 channels
  - 3x3 convolution; stride 1; valid padding; 16 channels
  - 3x3 max pooling; stride 2; valid padding;
  - 3x3 convolution; stride 1; valid padding; 32 channels
  - 3x3 max pooling; stride 2; valid padding;
  - 3x3 convolution; stride 1; same padding; 32 channels
  - 2 x FC layers

- **simple_model_2** (566k parameters)
  - 7x7 convolution; stride 1; valid padding; 16 channels
  - 7x7 convolution; stride 1; valid padding; 32 channels
  - 7x7 convolution; stride 1; valid padding; 64 channels
  - 5x5 convolution; stride 1; valid padding; 64 channels
  - 3x3 convolution; stride 1; valid padding; 128 channels

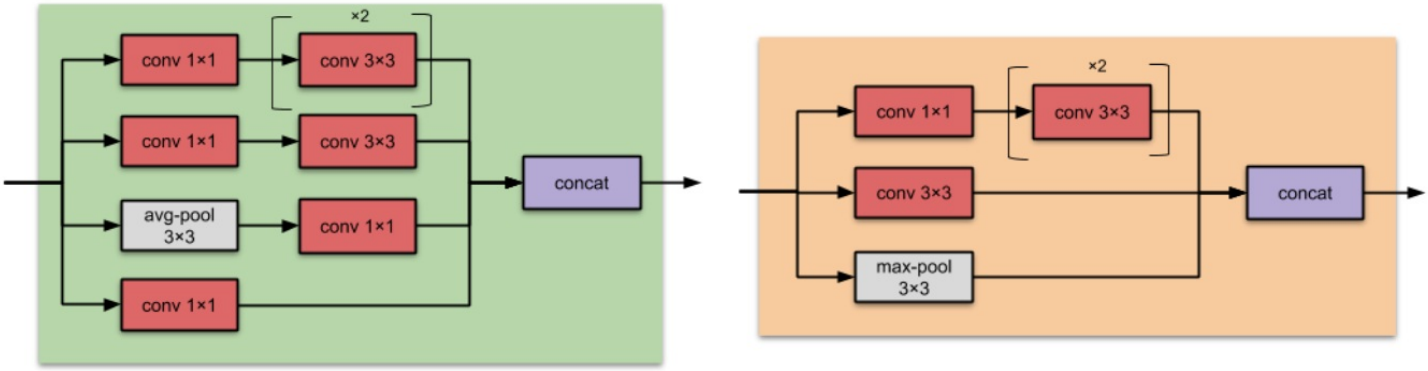
    - 2 x FC layers

- **inception** (343k parameters)
    - 3x3 convolution; stride 1; valid padding; 16 channels
    - 3x3 convolution; stride 1; valid padding; 16 channels
    - Inception-A block; 64 channels
    - Reduction-A block; 96 channels
    - Reduction-A block;
    - 1x1 convolution; stride 1; valid padding; 48 channels
    - 2 x FC layers

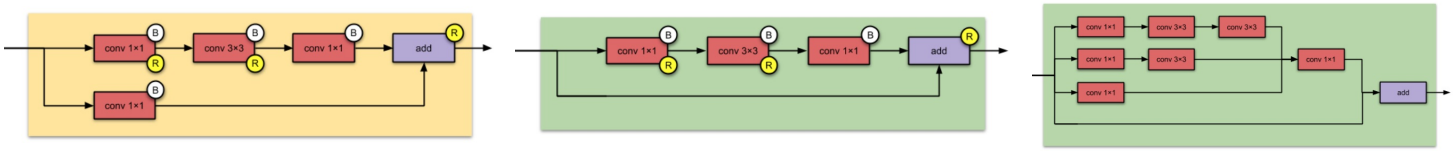
- **resnet** (1274k parameters)
    - 3x3 convolution; stride 1; valid padding; 16 channels
    - 3x3 convolution; stride 1; valid padding; 32 channels
    - Conv block; 64 channels
    - 2 x Identity block; 64 channels
    - 3x3 max pooling; stride 2; same padding
    - Conv block; 128 channels
    - 2 x Identity block; 128 channels
    - 3x3 max pooling; stride 2; same padding
    - 2 x FC layer

- **inception_resnet** (982k parameters)
    - 3x3 convolution; stride 1; valid padding; 16 channels
    - 3x3 convolution; stride 1; valid padding; 32 channels
    - 2 x Inception-ResNet block; 64 channels
    - Reduction-A block; 96 channels
    - 2 x Inception-ResNet block; 96 channels
    - Reduction-A block; 96 channels
    - 2 x FC layers

Inception-A block **Reduction-A block**



Conv block **Identity block** **Inception-ResNet block**



Images taken from: https://towardsdatascience.com/illustrated-10-cnn-architectures-95d78ace614d#6872 (https://towardsdatascience.com/illustrated-10-cnn-architectures-95d78ace614d#6872)

3.1. Metrics and optimization

For all models we use softmax cross-entropy loss for parameters optimization. Optimization algorithm employed is Adam with exponential learning rate decay. For majority models, initial value for learning rate is 5e-4, decresing coefficient is set to 0.9 and decreasing is done after 10000 steps (models specific parameters can be found in *config.json* file in model's output directory: *output_dir/model_dir*). As our aim is to optimize accuracy, that metric will be used as decision criteria during training. In other words, once we spot accuracy starts to decreasing or gets saturated on validation set we will stop training. For detailed analysis of model performance we also plot accuracy within each of 10 classes. This can be particularly helpful in pipointing model's weak points (for example mixing similar classes like T-shirt and Shirt).

4. Experiments

4.1. Comparing Architectures

First experiment involves comparing relative performance of different architectures. In order to isolate the effect of each architecture we keep all hyper-parameter fixed for all models (batch size, data augmentation options, regularization, learning rate, ...). In the following table can be found accuracy rate during training phase for 5 architectures (accuracy is given in percent and it is computed on validation set!). Besides overall accuracy rate we also present individual accuracy rate within several classes.

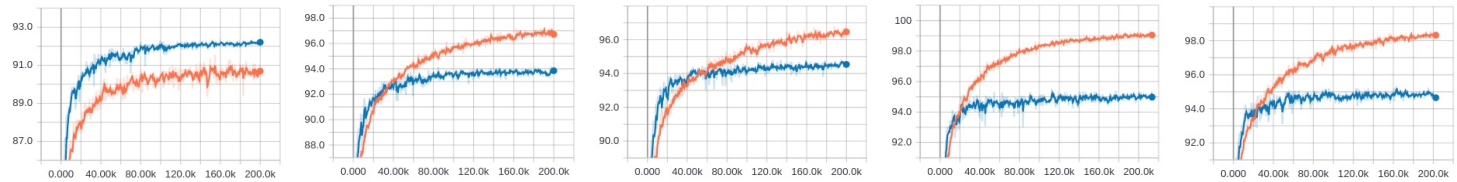
metric	Simple Model 1	Simple Model 2	Inception	ResNet	Inception-ResNet
Overall	92.15	93.7	94.5	**95.0**	94.8
T-shirt (class 0)	89.6	**91.0**	90.8	**91.0**	90.75
Trouser (class 1)	98.5	98.85	99.0	99.15	**99.25**
Shirt (class 6)	74.75	81.0	83.0	83.5	**83.75**

Here are some observations:

- Not surprisingly, more advanced architectures perform better than simplistic ones.
- Even very simple models can achieve descent results. Even more, simple_model_2 have comparable results with advanced architectures which leave space that proper choice of convolutions, number of layers and number of channels of VGG-like models can perform on par with advanced architectures.
- More advanced architectures significantly outperform simple models on more challenging classes like T-shirt and Shirt. On less challenging classes like Trouser difference is much smaller. Detailed performance per classes can be seen using Tensorboard visualisation.
- Note! We cannot conclude that last 3 models outperform simple ones just because of their more advanced architecture. Have in mind that these models have more layers and parameters.

Now, let's take a look into overall accuracy rate during training process for these 5 models:

Simple Model 1 **Simple Model 2** **Inception** **ResNet** **Inception-ResNet**



We observe that accuracy on validation set is higher than accuracy on training set for simple_model_1 throughout entire training process. This, not so usual behaviour, can be explained by the fact that training set is much more difficult to be learned than validation set (due to augmentation methods) combined with fairly simple model that is not capable of learning (and overfitting) more complex training set. Similar behaviour (due to augmented trainig set) can be also seen in the initial phases of all other methods where validation accuracy is higher than training accuracy. However, after more or less training steps these models begin to overcome noise introduced by augmentation and in later stages slightly overfit training data which is expressed in higher training accuracy as usual.

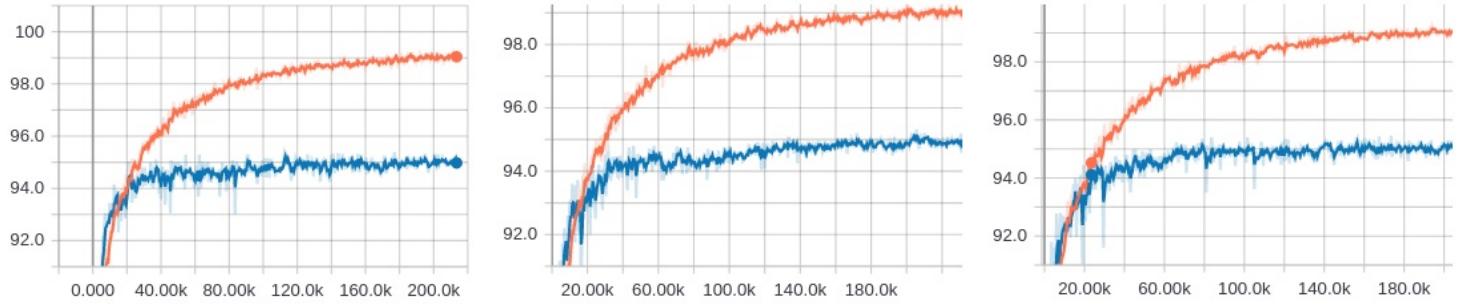
4.2. Comparing Loss Functions

There are several classes which members can be difficult to distinguish even by humans (T-shirt or Shirt, Shirt or Dress, etc.). All models consistently perform worse on such classes (accuracy ranges between 85 and 92 percents). To boost performance on these challenging classes we tried out weighted cross entropy loss. We denote following classes as challanging T-shirt, Shirt, Cout and Pullover and multiply their corresponding loss terms by a value of 2.

Additionally we tried out so called 'Focal loss' which basic idea is to down-weight the loss assigned to well-classified examples. Though, it is originally designed to tackle imbalanced dataset problem we hypothesize that it might have positive effects on our probelem where instead of imbalanced dataset we have more and less similar classes. Detailed description of Focal loss can be found in [this paper (https://arxiv.org/pdf/1708.02002.pdf)](https://arxiv.org/pdf/1708.02002.pdf).

On the following graphs accuracy rate during trainig is showed for ResNet model using 3 different loss functions. Though, significant change hasn't been observed, Focal loss (95.05 accuracy on validation set) outperformed cross-entropy loss (95.0) by a small margin while weighted cross-entropy (94.85) performed worse than cross-entropy loss.

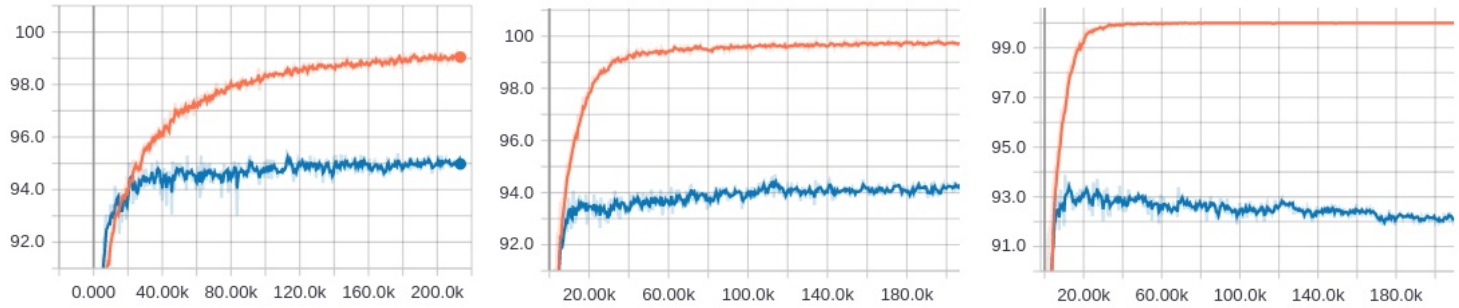
Cross-Entropy **Weighted Cross-Entropy** **Focal**



4.3. Data Augmentation Impact

To investigate impact of augmentation methods we trained 3 ResNet models with different augmentation probability. We used value of 0.3, 0.05 and 0 meaning no augmentation at all. In the following graphs accuracy rate during training is shown.

Augmentation probability: 0.3	Augmentation probability: 0.05	No augmentation



Obviously augmentation methods we applied have great impact on model's performance and have strong regularization effects. Ommiting augmentation leads to overfitting training data very quickly (accuracy on training set is 1 one after only 30-40k steps) and performing poorely on validation set. Moderate augmentation probability (0.05) prevents model from learning training set 'by heart' but still doesn't perform on the same level as more aggresive augmentation with probability of 0.3. Eventhough we concluded that more aggresive augmentation is better, it remains unclear if optimal value for augmentation probability is bigger or smaller than 0.3. For that reason we ran additional training of ResNet model with augmentation probability set to 0.5 and bigger weight decay as well. However, no improvement has been observed (result not shown).

We haven't done any experiments to investigate relative performance of each augmentation method.

5. Final Model

Since we haven't observed drop in validation accuracy during training process after pleteauing of both training and validation accuracy (except in the case where regularization hasn't been used) we can be sure with reasonable probability that we won't enter overfitting phase when training without validation set. That way we can use entire training set to fit parameters. Thus, for final model we use all 60000 examples for training and trained for 250000 steps. We use ResNet architecture with Focal loss. Other training parameters can be found in *output_dir/final_model/config.josn* file.

Path to the exported inference graph is *output_dir/final_model/frozen_inference_graph.pb*.

Performance on test data (10k examples) is given in the table bellow. To reproduce these results run *benchmark.py* script.

Class	Accuracy
Overall	94.23
Class 0	91.0
Class 1	99.2
Class 2	93.4
Class 3	93.9
Class 4	94.1
Class 5	98.6
Class 6	78.2
Class 7	98.0
Class 8	99.5
Class 9	96.4

This results tells us that we can expect to achieve accuracy around 94.2% on unseen examples created in the same fashion as fashion mnist t10k set.

6. Demo Program

We didn't use any detection algorithm to localize object for classification. Instead we enforce users of our small demo program to place desired object within the blue rectangle area once the program gets started. Program is implemented in *demo_program.py* script. Small demonstration how it works can be seen in *demo_video.avi* in the root directory.

7. Summary

To conclude, we managed to get results very close to current state of the art models on test set with not to deep model containing around 1.3M parameters. We realised that augmentation methods have great impact on end performance so further improvement should be directed in finding more siphisticated augmentation methods.

Here are some more ideas we come up with but weren't able to implement them:

- Try out CNN models with dense blocks.
- Try out incorporating triplet loss in objective function. This idea seems very reasonable since the model would focus on 'hard-to-distinguish' examples like T-shirt and Shirt.
- Perform more detailed search of hyperparameters related to network size like number of layers, number of inception/resnet blocks, etc.

In [ ]:

```

```

In [ ]:

```

```