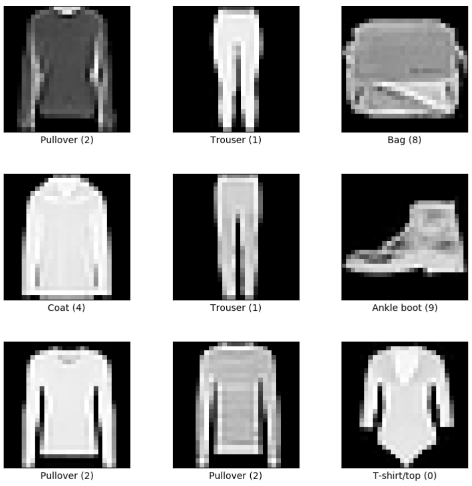
INFOMCV Assignment 4

G. Picarella (Group 34)

Dataset

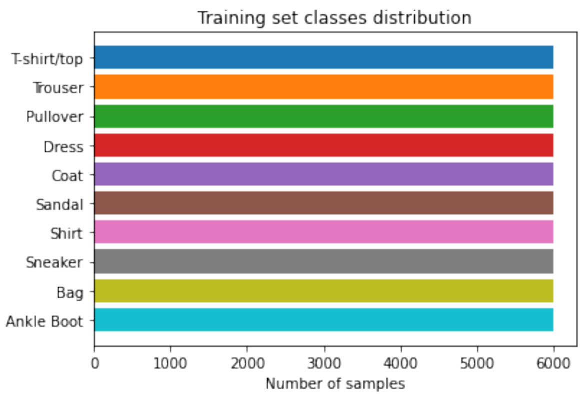
Description

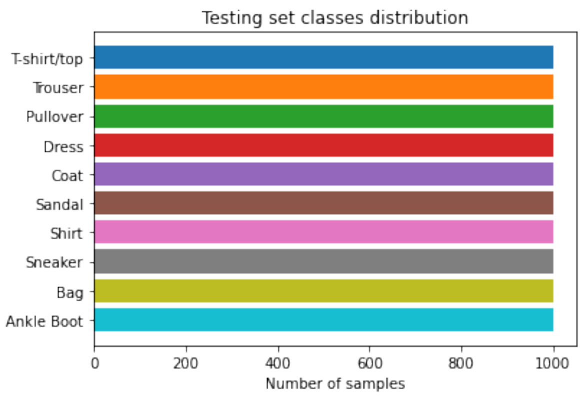
For this assignment, we are working with the Fashion MNIST dataset. The dataset was downloaded from Keras via the function keras.datasets.fashion_mnist.load_data(). Fashion-MNIST is a dataset containing images of clothes from the Zalando repository; the dataset consists of a training set of 60k samples and a test set of 10k samples. Each sample in the dataset is a 28x28x1 grayscale image of a piece of clothing, associated with a label referring to 10 different classes: T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, and Ankle Boot.



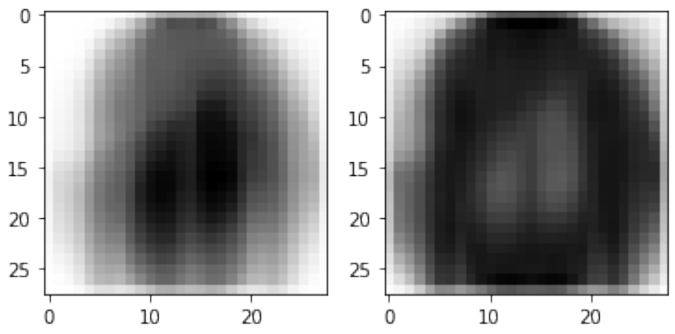
Analysis

Our CNNs architectures and performance metrics are based on the insights we were able to obtain from a quick data analysis over the dataset. The first investigation carried out involves class imbalances in our training and testing set. We collected and plotted each class' distribution in the dataset. The results show 10 balanced classes with 6000 and 1000 samples each for the training and testing set respectively. This insight affects our assignments in two main ways: we don't need to employ any data balancing strategy to our dataset and, most importantly, we can use accuracy as a valid and meaningful performance measure during our models evaluation.





Next, we carried out another investigation involving padding requirements for the samples in our dataset. We computed the mean and std for each pixel contained in the 28x28x1 grayscale shape of our input and checked the amount of information contained along the borders of the picture. As the std suggests, the majority of information can be found as we go towards the picture center. Nonetheless, without a minimum padding we would lose part of the information contained in the bottom border of the picture where valuable information may be stored. In our case, we use 5x5 kernels and thus we apply a 2px padding to all our samples.



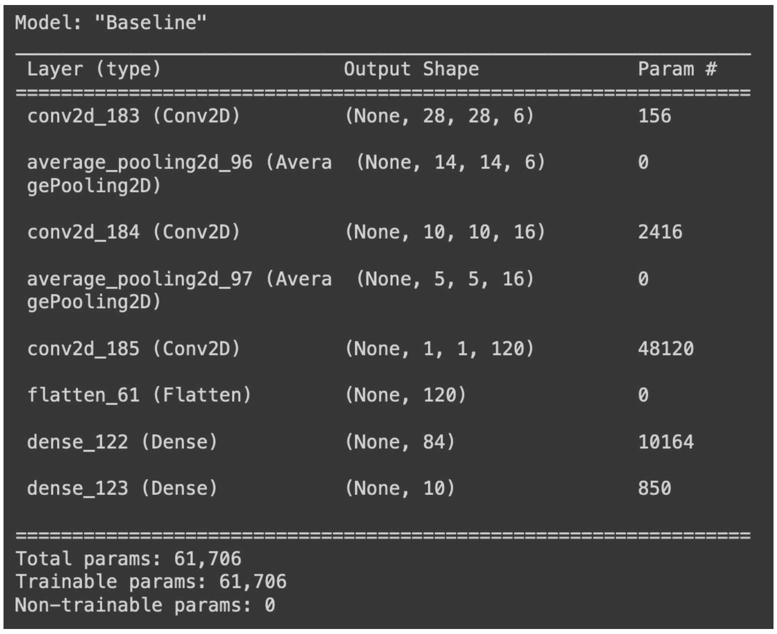
Preprocessing

The Fashion MNIST dataset doesn't require an extensive nor complex preprocessing. We normalised each grayscale pixel colour in the range [0, 1] for a better model stability during training. Next, we forced the size of each samples to be 28x28x1 by discarding any other additional dimension in each sample shape.

CNN Architectures

Baseline model

Our baseline model employs an architecture similar to LeNet-5. We decided to employ this architecture because of a series of reasons: first, the classification problem we are trying to solve and the one solved by the original LeNet-5 CNN (MNIST handwritten dataset) share quite some interesting features like the input size and colour space; second, given the >99% accuracy obtained with the MNIST dataset, we expect LeNet-5 to perform reasonably well also in this case; third, LeNet-5 is considered a small CNN by today's standard (61,706 trainable parameters in our version), thus allowing for faster training and trial and error phases during the assignment. Our baseline comprises 3 convolution layers, 2 average pool layers, 1 Flatten and 2 Dense layers with 84 and 10 neurons respectively. The model architecture can be broadly divided in visual feature extraction and classification stages. Our baseline model comprises 5 layers responsible for feature extraction and 3 layers aimed for classification. For each layer non-pooling layer, we decided to use the TanH activation function. This is because the derivatives of the TanH are larger than the derivatives of most other activation functions, thus typically the minimisation of the cost function during training is faster. Nonetheless, there are surely better activation functions (ReLu, ReLuLeaky) that we already employ for some of our proposed baseline variations and which we will discuss next. The first convolution layer adds a padding of 2 pixels (28x28x1 to 32x32x1) and applies 6 5x5 kernels to the input sample, thus producing a 28x28x6x1 activation volume in output. The layer uses a stride set to 1, so the receptive field of each element in the activation volume shares part of the neighbours in the input sample with its neighbours. We use a padding equal to 2 for each side of the input sample, so that we don't miss any information stored at the border during the convolution. This layer includes 156 trainable parameters, 26 for each kernel (25 weights + 1 bias). The second average pooling layer uses a kernel extent and stride set to 2, thus shrinking the activation volume by a factor of 2 along the 28x28 dimensions. Consequently to this operation, each activation map is smoothed and a 14x14x6x1 output activation volume with a 2x2 non-overlapping receptive field is produced. The third convolution layer shares all the settings used in the first layer, except for the number of kernels (set to 16) and padding (set to 0) used. The layer produces a 10x10x16x1 activation volume in output and includes 2416 trainable parameters, 26 for each kernel (25 weights + 1 bias). The fourth average pooling layer shares all the settings used in the second layer. It takes a 10x10x16x1 volume in input and produces a smaller 5x5x16x1 activation volume in output. The fifth convolution layer shares all the settings used in the first layer, except for the number of kernels used (set to 120). Given that the fourth layer produces a 5x5x16x1 activation volume and that the current layer uses 5x5 kernels, the produced output activation volume will be of size 120x1x1. This layer includes 48120 trainable parameters, 26 for each kernel (25 weights + 1 bias). The sixth flatten layer takes the 120x1x1 volume produced by the previous convolution layer and flattens it to a 1D vector. The seventh dense layer is composed by 84 neurons, each of which takes 120 values plus 1 bias in input from the previous layer. This means that 10164 trainable parameters are required for this layer. Finally, the activation from the seventh layer is connected to the final dense output layer. Our output layer contains 10 neurons (one for each class) and estimates the expected multinomial probability distribution for each class through a Softmax activation function.

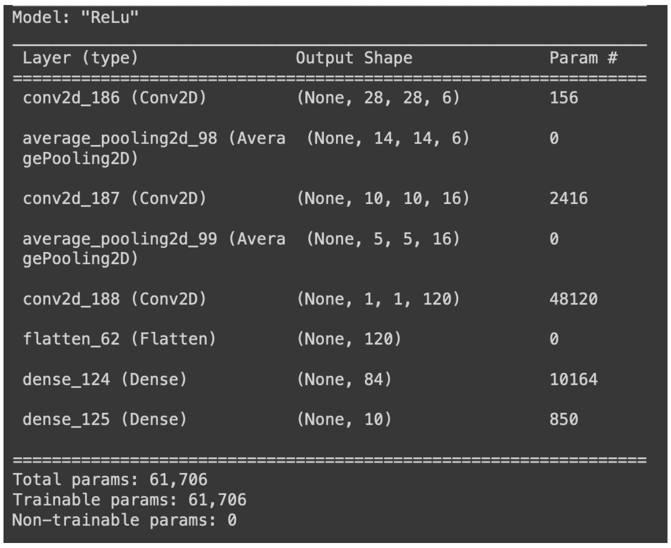


```
Model: "Baseline"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_183 (Conv2D)         (None, 28, 28, 6)         156

 average_pooling2d_96 (Avera  (None, 14, 14, 6)        0
 gePooling2D)

 conv2d_184 (Conv2D)         (None, 10, 10, 16)        2416

 average_pooling2d_97 (Avera  (None, 5, 5, 16)         0
 gePooling2D)

 conv2d_185 (Conv2D)         (None, 1, 1, 120)         48120

 flatten_61 (Flatten)        (None, 120)               0

 dense_122 (Dense)           (None, 84)                10164

 dense_123 (Dense)           (None, 10)                850

=================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
```

ReLU variation

Our first baseline variation employs the ReLU activation function in every layer that was previously using the TanH activation function. The reason why we decided to use ReLu for this variation is two-fold: first, ReLU guarantees a much lower likelihood for the gradient to vanish (constant gradient), thus resulting in faster cost function minimisation during training; second, ReLUs are more sparse when the activation value is lower than or equal to 0;

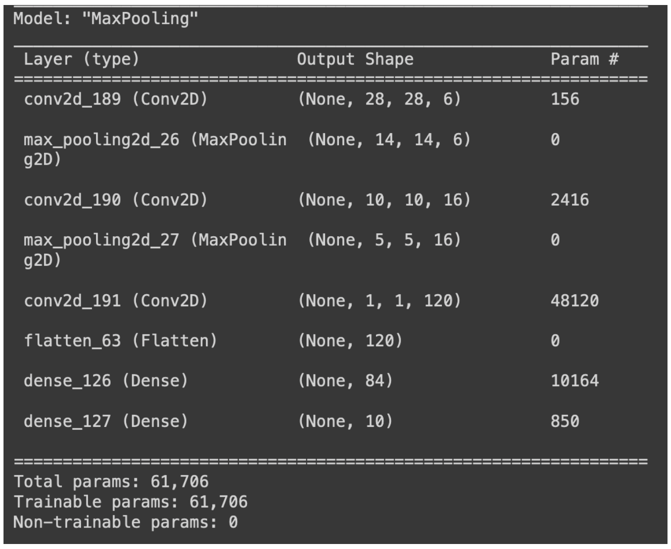
generally, sparse representations seem to give better results than dense representations. There is also an additional reason why ReLU should be used, which is its computation efficiency (only a comparison, addition and multiplication is required) when compared with TanH or other types of Sigmoid functions employing exponentials and divisions. This last property should also speed-up the training phase when dealing with a good amount of data or limited computational resources.



```
Model: "ReLu"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_186 (Conv2D)         (None, 28, 28, 6)         156

 average_pooling2d_98 (Avera  (None, 14, 14, 6)        0
 gePooling2D)

 conv2d_187 (Conv2D)         (None, 10, 10, 16)        2416

 average_pooling2d_99 (Avera  (None, 5, 5, 16)         0
 gePooling2D)

 conv2d_188 (Conv2D)         (None, 1, 1, 120)         48120

 flatten_62 (Flatten)        (None, 120)               0

 dense_124 (Dense)           (None, 84)                10164

 dense_125 (Dense)           (None, 10)                850

=================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
```

MaxPooling variation

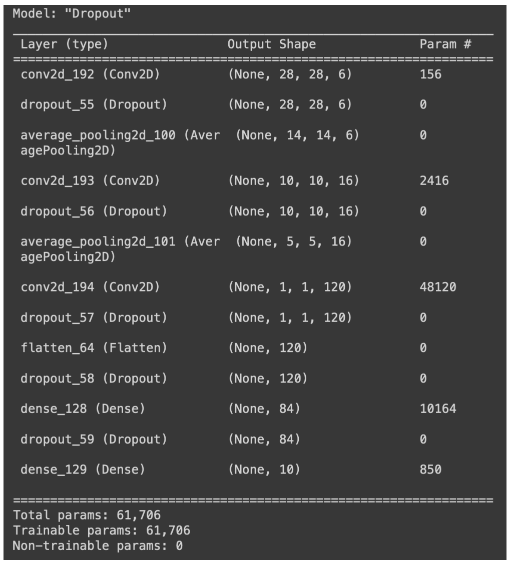
Our second baseline variation replaces each average pooling layer with a max pooling layer with same extent and stride. The reason why we decided to change the pooling operation is because sometimes average pooling struggles extracting important features from the patch because it takes every element into account (i.e. reduces overall contrast), giving an average which might take into consideration irrelevant features. On the other hand, max pooling only focuses on maximum activation (a completely opposite approach), thus we think it's reasonable and worthwhile to try.



```
Model: "MaxPooling"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_189 (Conv2D)         (None, 28, 28, 6)         156

 max_pooling2d_26 (MaxPoolin  (None, 14, 14, 6)        0
 g2D)

 conv2d_190 (Conv2D)         (None, 10, 10, 16)        2416

 max_pooling2d_27 (MaxPoolin  (None, 5, 5, 16)         0
 g2D)

 conv2d_191 (Conv2D)         (None, 1, 1, 120)         48120

 flatten_63 (Flatten)        (None, 120)               0

 dense_126 (Dense)           (None, 84)                10164

 dense_127 (Dense)           (None, 10)                850

=================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
```

Dropout variation

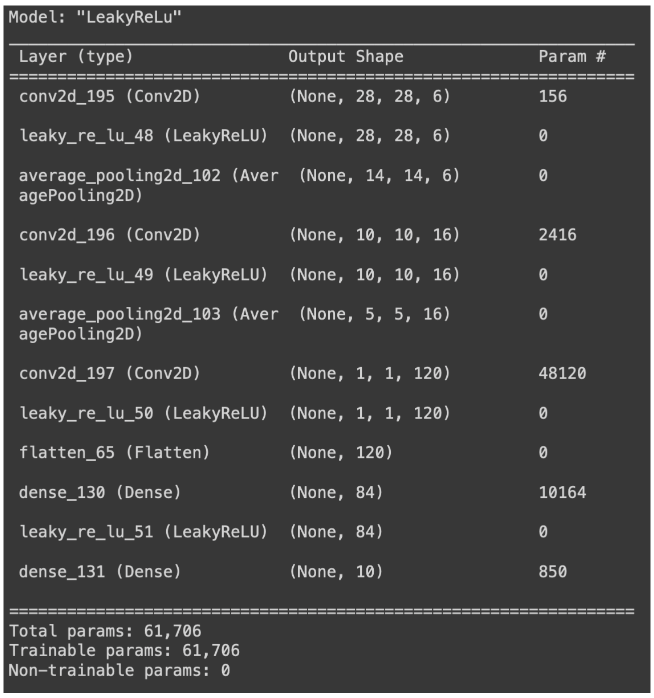
Our third baseline variation implies 4 additional dropout regularisation layers (the first 3 with p = 0.25 and the last one with p = 0.5) on top of the base architecture. Based on the dropout probability, a dropout layer has the effect of temporary ignoring a subset of the inputs for a particular neuron in the next layer. Each dropout layer is inserted after a convolution or dense layer and its activation function application. For the convolution layers, we keep the dropout rate low (0.25) because the number of connections is already limited. In contrast, for the dense layer we



```
Model: "Dropout"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_192 (Conv2D)         (None, 28, 28, 6)         156

 dropout_55 (Dropout)        (None, 28, 28, 6)         0

 average_pooling2d_100 (Aver  (None, 14, 14, 6)        0
 agePooling2D)

 conv2d_193 (Conv2D)         (None, 10, 10, 16)        2416

 dropout_56 (Dropout)        (None, 10, 10, 16)        0

 average_pooling2d_101 (Aver  (None, 5, 5, 16)         0
 agePooling2D)

 conv2d_194 (Conv2D)         (None, 1, 1, 120)         48120

 dropout_57 (Dropout)        (None, 1, 1, 120)         0

 flatten_64 (Flatten)        (None, 120)               0

 dropout_58 (Dropout)        (None, 120)               0

 dense_128 (Dense)           (None, 84)                10164

 dropout_59 (Dropout)        (None, 84)                0

 dense_129 (Dense)           (None, 10)                850

=================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
```

increase the dropout rate to the standard value suggested in literature (0.5). The reason why we decided to use dropout layers is because generally its behaviour improves the robustness and resilience of the network while also preventing overfitting.

LeakyReLU variation

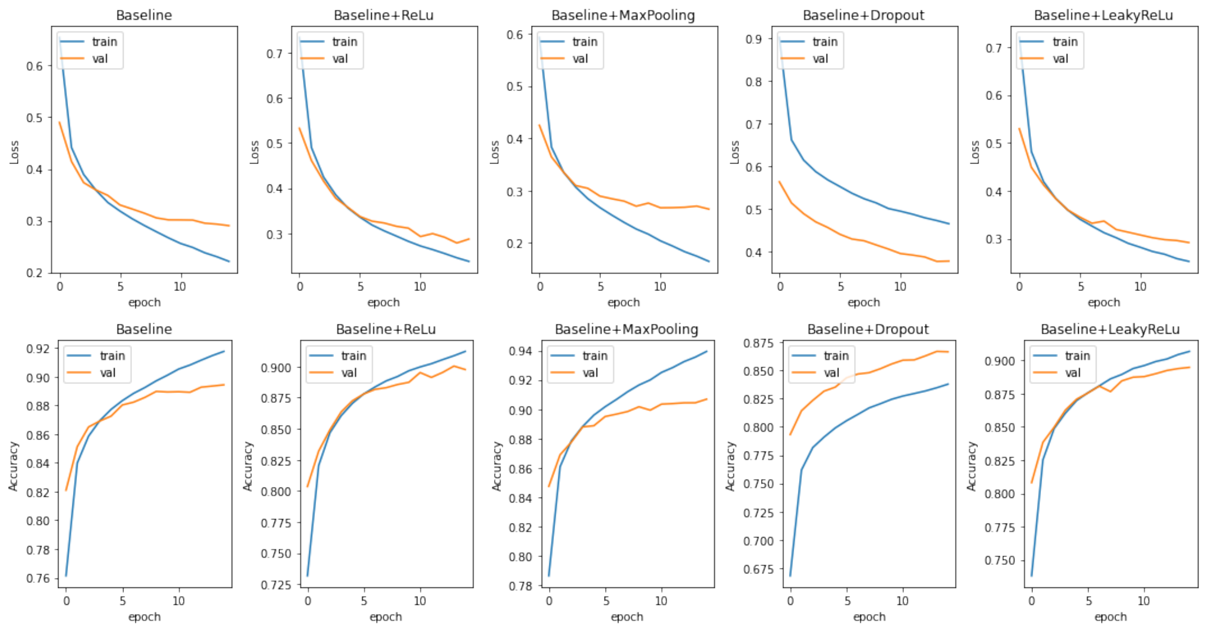


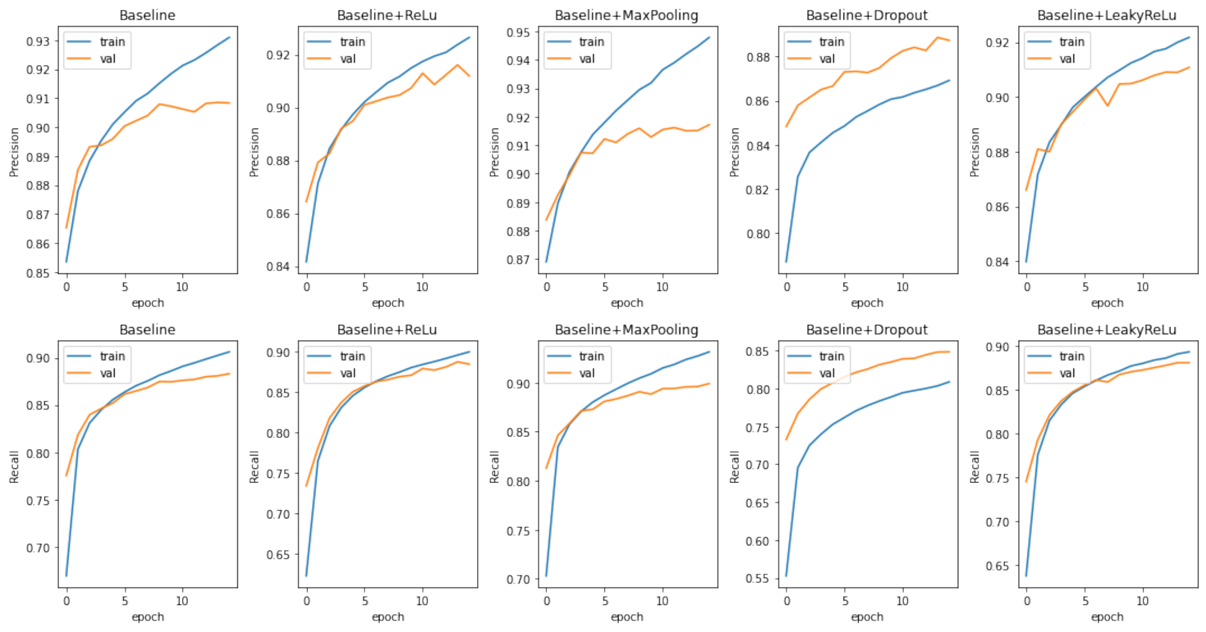
```
Model: "LeakyReLu"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_195 (Conv2D)         (None, 28, 28, 6)         156

 leaky_re_lu_48 (LeakyReLU)  (None, 28, 28, 6)         0

 average_pooling2d_102 (Aver  (None, 14, 14, 6)        0
 agePooling2D)

 conv2d_196 (Conv2D)         (None, 10, 10, 16)        2416

 leaky_re_lu_49 (LeakyReLU)  (None, 10, 10, 16)        0

 average_pooling2d_103 (Aver  (None, 5, 5, 16)         0
 agePooling2D)

 conv2d_197 (Conv2D)         (None, 1, 1, 120)         48120

 leaky_re_lu_50 (LeakyReLU)  (None, 1, 1, 120)         0

 flatten_65 (Flatten)        (None, 120)               0

 dense_130 (Dense)           (None, 84)                10164

 leaky_re_lu_51 (LeakyReLU)  (None, 84)                0

 dense_131 (Dense)           (None, 10)                850

=================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
```

Our fourth baseline variation employs the LeakyReLU activation function in every layer that was previously using the TanH activation function. The reason why we decided to use LeakyReLU for this variation is because it brings to the network all the improvements obtained with ReLU, while solving some of the drawbacks that arise with it. In particular, large weight updates in the network may result in a summed input passed to the activation function below zero, producing an output value of 0. This behaviour may result in large dead areas of the network where neurons do not add any contribution to the classification because their output is always 0 (also called *dying ReLU* state). LeakyReLU addresses this issue by allowing small negative values when the summed input is less than zero.

Training and validation loss, accuracy, precision and recall for all five models

All the trainings are performed with the Adam optimiser and the categorical cross-entropy loss function.





[Link to our models' weights]()

Table with training and validation top-1 accuracy for all five models

Model name	Training top-1 accuracy (%)	Validation top-1 accuracy (%)
Baseline	0.9205@fold 0	0.8989@fold 1
ReLU	0.9141@fold 3	0.9054@fold 3
MaxPooling	0.9405@fold 0	0.9115@fold 1
Dropout	0.8406@fold 1	0.0.8728@fold 0
LeakyReLU	0.9084@fold 1	0.8999@fold 3

Discuss your results in terms of your model

Baseline vs ReLU

The comparison of the two loss graphs shows that the ReLU variant generalised the classification task much better than the baseline model (lower validation loss). This behaviour can be explained by considering the dying ReLU issue: when using ReLU, if the summed input of the activation function of a neuron is negative, then there is a high chance that its output value will be fixed to 0 forever, thus no longer providing any contribution to the network. By some extent, the dying ReLU effect can be considered equivalent to what a dropout layer is supposed to do, i.e. disabling some of the inputs of the neurons in the next layer with a specific dropout probability. Nonetheless, the effect we get with a dropout layer is much more evident because it's specifically tailored for that while, in this case, the dying ReLU issue makes this behaviour less noticeable. On the other hand, the training curves do not show interesting arguments of discussion; they show a pretty good training loss minimisation for both models. For what concerns the validation accuracy, precision and recall, our ReLU variant achieves higher results in all the performance metrics.

Model name	**Accuracy**	**Precision**	**Recall**	**Loss**
Baseline	0.8908	0.9049	0.8802	0.2952
ReLU	0.8955	0.9098	0.8809	0.2917

Baseline vs LeakyReLU

As in the ReLU variant, we see that the validation loss curve is lower for our ReLULeaky variant. Again, this behaviour can be explained by the dying ReLU issue. We noticed a validation loss increase of 0.03 between our ReLU and LeakyReLU variants: this can be explained by the fact that ReLULeaky aims at minimising the dying ReLU issue by allowing small negative values when the summed input is less than zero. For what concerns the validation accuracy, precision and recall, our ReLULeaky variant achieves higher results in all the performance metrics except recall.

Model name	**Accuracy**	**Precision**	**Recall**	**Loss**
Baseline	0.8908	0.9049	0.8802	0.2952
LeakyReLU	0.8964	0.9147	0.8812	0.2670

Baseline vs Dropout

The dropout layers show a similar behaviour as with the dying ReLU issue but, as we said beforehand, it's much more evident what is going on here: each dropout layer is preventing overfitting and increasing the model reliability and robustness by setting some of the neurons inputs to 0 with a probability of 0.25 and 0.5 based on the specific layer. The validation loss curve is bounded by the training loss curve for the entire graph. We also notice a slower convergence rate for the dropout variation: this is due to the fact that the convergence speed is affected by the neurons' inputs set to 0 by the dropout layer, thus limiting the flow of information in the network. A possible mitigation for this issue would be increasing the number of epochs during training, but unfortunately, we were not allowed to use more than 15 epochs for this assignment. For what concerns the validation accuracy, precision and recall, our baseline model achieves higher results in all the performance metrics. This can be explained by the faster convergence rate achieved by our baseline model during the training phase.

Model name	**Accuracy**	**Precision**	**Recall**	**Loss**
Baseline	0.8908	0.9049	0.8802	0.2952
Dropout	0.8665	0.8904	0.8439	0.3779

Baseline vs MaxPooling

The comparison of the two loss graphs show that our max pooling variation achieves a better validation loss while minimising the training loss faster. Nonetheless, we also notice that, among all models, it's the one with the highest overfitting. It seems like the maximum activation selected for each patch by the max pooling operation is producing visual features which don't generalise well for our classification task. For what concerns the validation accuracy, precision and recall, our max pooling model achieves higher results in all the performance metrics.

Model name	**Accuracy**	**Precision**	**Recall**	**Loss**
Baseline	0.8908	0.9049	0.8802	0.2952
MaxPooling	0.9068	0.9172	0.8995	0.2594

Best performing models

Based on the validation accuracy metric, we selected ReLU and LeakyReLU as our best two models. We didn't consider MaxPooling in the choice because it was overfitting much more than the other models and it would have been surely an issue in the testing phase.

Discuss the differences between the two models evaluated on the test set

We trained again our models on the entire training set and then evaluated their performance metrics using the entire testing set. The results show similar metrics, even though the ReLU model has slightly better values for each metric. As we previously mentioned, the dying ReLU issue turned to be beneficial for the network in this case. Surely this is highly correlated to the limited size of our baseline model, the optimiser used, the learning rate and the classification problem we are trying to solve. It's difficult to say what could really turn the dying ReLU issue in a real bottleneck for this network; additional investigations are surely needed. For completeness, we report the test loss, accuracy, precision and recall for each of our best models.

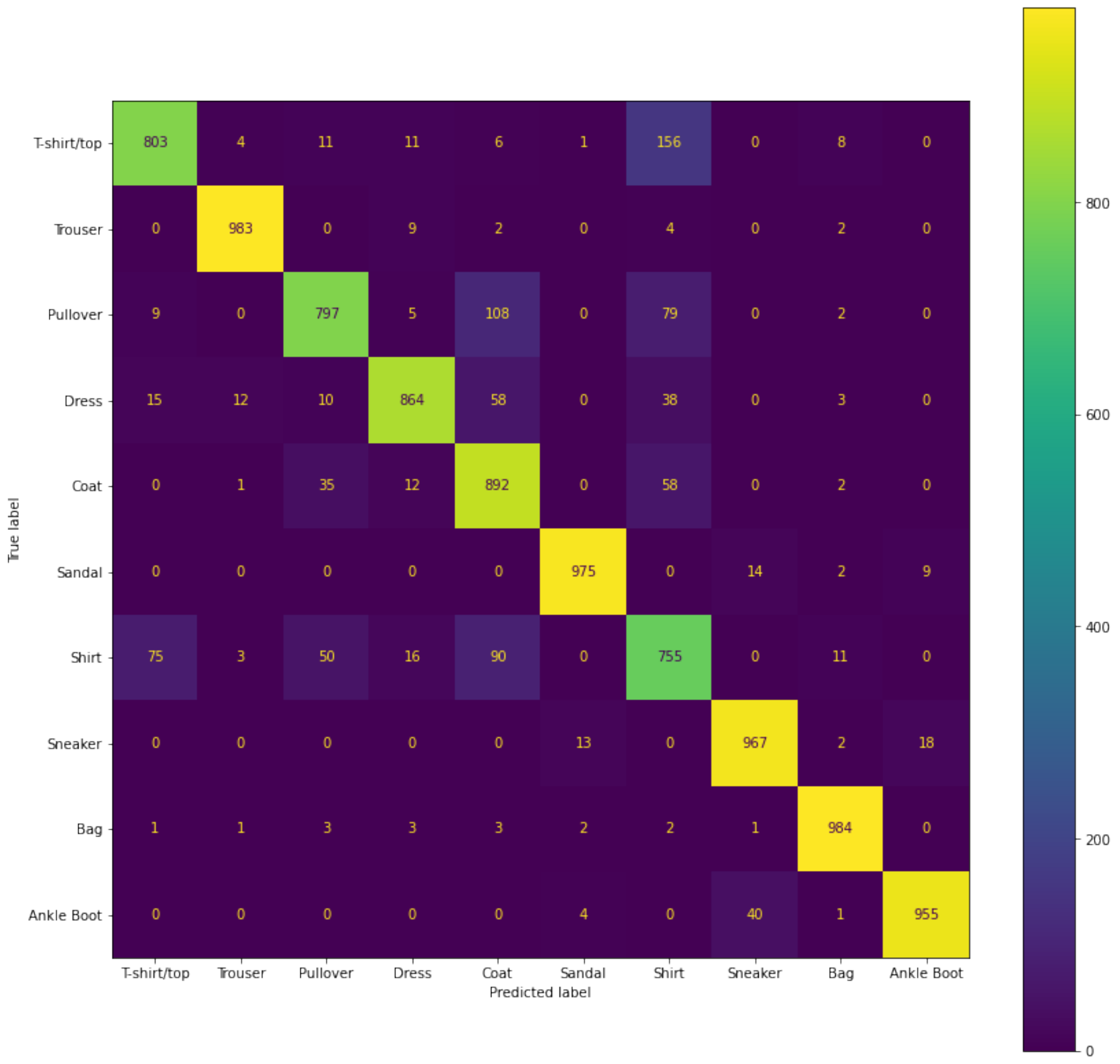
Model name	**Accuracy**	**Precision**	**Recall**	**Loss**
ReLU	0.8974	0.9089	0.8880	0.2836
LeakyReLU	0.8909	0.9083	0.8761	0.3008

Our performance parameters show that both in the validation and test phase the ReLU variation yields better results when compared with the LeakyReLU variation. Thus, the validation parameters provided a realistic outline of the generalisation capabilities of both models.

Choice tasks

CHOICE 1: Provide and explain a confusion matrix for the results on the test set of one of the models

We analyse the confusion matrix obtained with the ReLU variant. As expected, most of the values are concentrated along the main diagonal. The majority of misclassifications happen between similar classes: among the most noticeable ones, we have 156 samples in the T-shirt/top class wrongly classified as Shirt, while we have 75 samples in the Shirt class wrongly classified as T-Shirt/top. This misclassification pattern is shared among all the classes pairs which present visual similarities in their samples. We believe that this problem can be addressed in multiple ways: by using data augmentation techniques, with more detailed samples (i.e higher resolution), different learning settings (i.e optimisers, learning rate) or a different model architecture. On the other hand, we have a high classification accuracy with respect to classes whose samples do not share substantial visual similarities with other classes (like Bags, Sneakers and Trousers).



	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle Boot
T-shirt/top	803	4	11	11	6	1	156	0	8	0
Trouser	0	983	0	9	2	0	4	0	2	0
Pullover	9	0	797	5	108	0	79	0	2	0
Dress	15	12	10	864	58	0	38	0	3	0
Coat	0	1	35	12	892	0	58	0	2	0
Sandal	0	0	0	0	0	975	0	14	2	9
Shirt	75	3	50	16	90	0	755	0	11	0
Sneaker	0	0	0	0	0	13	0	967	2	18
Bag	1	1	3	3	3	2	2	1	984	0
Ankle Boot	0	0	0	0	0	4	0	40	1	955

CHOICE 2: Create and apply a function to decrease the learning rate at a 1/2 of the value every 5 epochs

We decrease the learning rate by a factor 0.5 every 5 epochs with the function learning_rate_step_decay passed as a step callback to model.fit(…). Then we trained our best performing model (ReLU variant) with the learning rate's decay strategy enabled. The performance metrics derived from the evaluation on the testing set shows an overall improvement of the model, with loss, accuracy, precision and recall.

Model name	**Accuracy**	**Precision**	**Recall**	**Loss**
ReLU	0.9085	0.9167	0.9016	0.3028

These results can be explained by considering two main facts: first, a smaller learning rate allows the optimiser to take smaller steps towards the minima and hence doesn't skip the minima so easily; second, a smaller learning rate also limits the dying ReLU issue by some extent, because big weight values jumps are less common (ReLU has also a constant gradient, 0 if the input value is lower than or equal to 0 and 1 otherwise).

CHOICE 3: Instead of having a fixed validation set, implement k-fold cross-validation

We use a stratified implementation of k-fold cross-validation which preserves each class distribution when splitting the data in training and validation sets. Our k-fold split is implemented in the sklearn library StratifiedKFold(…).split(…). Each aggregate metric for the trained model is computed as follows: for each fold, the history returned by model.fit(…) is stored in a list; once the training phase is done, the aggregate history dictionary is computed by dividing the sum of all dictionaries by the number of splits. We employ stratified 10-fold cross validation during each model training in the best two models' selection phase because it provides a more realistic estimate of the model's generalisation performance. Given the fairly big size of our training set (60k samples) and the fact that we use every chunk of our training set when computing the validation metrics, we decided to use k = 5 so that a 80%-20% split for training and validation at each step is used. This way we ensure that our model has sufficient data for the training phase and sufficient variance in the data for the validation metrics computation.