1. Purpose and data preparation

The purpose of this homework is to train a CNN shaped like the **AlexNet** on the **Caltech-101** dataset.

AlexNet is an 8-layer CNN, whose first five layers are convolutional and the last three are fully connected. The total number of weights to be optimized is roughly 60M, thus requiring a huge dataset to be properly trained. The only change we had to make was in the last FC layer, which we need to have 101 output instead of 1000.

By default, AlexNet implements a 0.5-Dropout for the FC layers and some Max-Pooling layers with re-LU after some of the convolutional layers (1, 2 and 5).

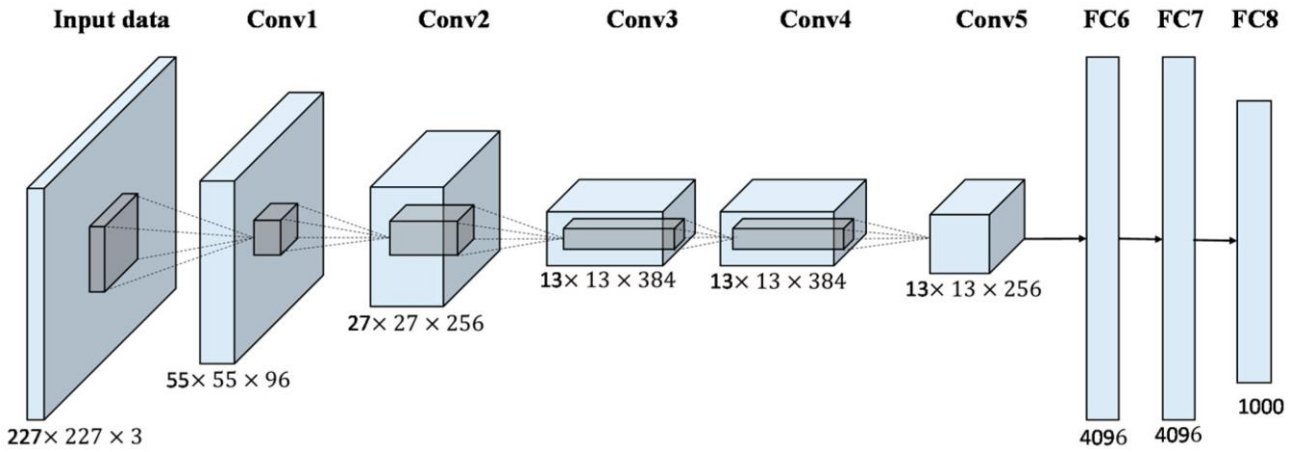


Figure 1 - The original AlexNet architecture

Unfortunately, the Caltech-101 dataset only contains about 9.000 images for 101 categories (102 actually, but the "BACKGROUND" class have been filtered out), most of which have about 50 images. Moreover, we must evenly divide the data into test, train and validation: we expect the results obtained training the whole network from scratch to be quite poor. Nevertheless, the objective of our analysis is to better understand how different parameters affect the behavior of the network rather than obtaining a super performing classifier.

While the division between train and test has already been provided, we need to split the training data into actual training and validation. We decide to keep a 1:1 ratio between the two, and to force the distribution of the categories to be even, we exploit the *stratify* parameter of the *train_test_split* function provided by the *sklearn* library.

2. Training from scratch

The first thing we will try to do is the training from scratch. This means that all the weights are initialized randomly and through backpropagation based on stochastic gradient descent the system will try to update the weights to optimize the cross-entropy loss function, defined as follows for a single classification:

$$\sum_{c=1}^{C} y_{o,c} \log(p_{o,c})$$

where *C* is the number of the classes, *y* is a binary indicator indicating if class label *c* is correct for observation *o*, and p is the predicted probability of observation *o* to belong to class *c*.

The parameters of the network which we will focus on are the following:

- *BATCH_SIZE*: the size of the batches for the training phase;
- *LR*: the learning rate;
- *NUM_EPOCHS*: the number of epochs which the training will be run for;
- *GAMMA*: the multiplicative factor for the learning rate step-down;
- *STEP_SIZE*: number of epochs after which the LR is updated using the gamma parameter through a step-down policy;

In this phase we decided to keep the BATCH_SIZE fixed (256 images per batch), while varying the other parameters. To estimate the goodness of the model, after each epoch the model itself is validated on both the training and the validation set, while the loss is computed and stored after each step of the learning phase (after each batch). Only the state of the model getting the best result in terms of accuracy on the validation set in then used to make the predictions on the test set.

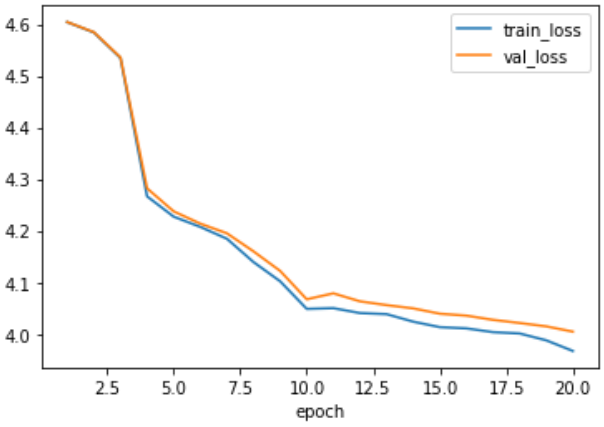
The table below shows the results of the different runs.

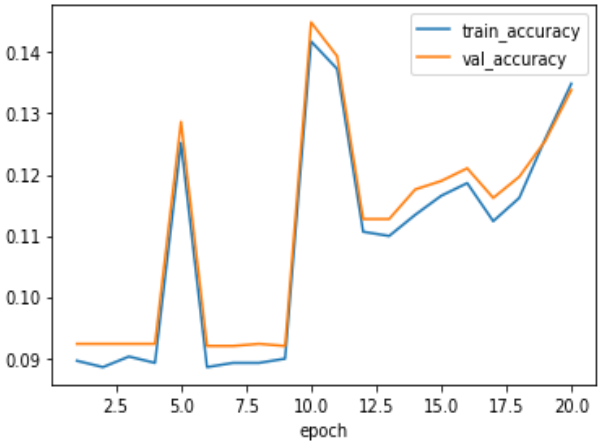
LR	NUM_EPOCHS	STEP_SIZE	GAMMA	MAX_VAL_ACC	TEST_ACC
0.001	30	20	0.1	0.16	0.09
0.005	20	10	0.1	0.15	0.13
0.01	30	15	0.1	0.25	0.26
0.05	**30**	**20**	**0.1**	**0.47**	**0.49**
0.1	30	5	0.5	0.40	0.42
0.05	40	20	0.1	0.41	0.42

As we can see, training the network from scratch requires a quite high learning rate, and even in the best case, due to the limited size of the dataset, we are not able to get above 50% of accuracy on the evaluation set even if we increase the number of epochs.

To better understand what happens during the training phase, let's inspect the behavior of the loss and the accuracy on training and validation set both when the network produces "acceptable" results and when it works poorly.

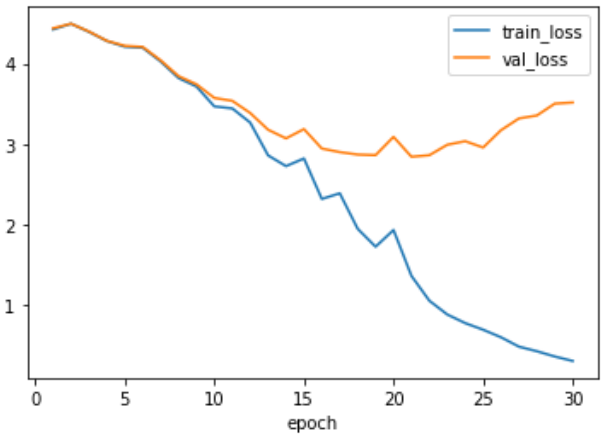
- LR = 0.005, NUM_EPOCHS = 20, STEP_SIZE = 10, GAMMA = 0.1

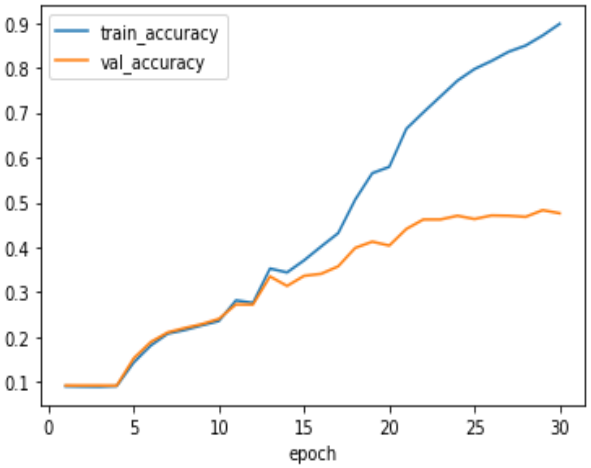




When the *LR* is too small, the network is not able to learn enough from the data: even if the loss shows a decreasing trend, the difference between the starting value and the final one is about 10%, and after some epochs it stops decreasingpdf. The result on the evaluation reflects this behavior, showing no clear upward trend as a good training would.

- LR = 0.05, NUM_EPOCHS = 30, STEP_SIZE = 20, GAMMA = 0.1





Increasing the *LR* of a x10 factor, all the previously described problem are solved, even though the accuracy on the validation saturates at roughly 0.47, while the training accuracy keeps increasing: this reveals a strong overfitting component, which unfortunately could only be solved by having more data to train the network. The other solution typically used to decrease overfitting, the dropout, is in fact already performed by AlexNet by default.

In section 4 of this report we'll try to overcome the overfitting by synthetically increasing the size of the dataset thanks to some data-augmentation techniques.

3. Transfer Learning

As shown in the previous section, training the model from scratch with a limited amount of data can lead to nothing more than a poor classifier. One technique that can be used in such a case is the *Transfer Learning*: the starting state of the network isn't random as before, instead it comes from a previous training already performed on the same network. In our case, the starting point is the AlexNet pre-trained on the ImageNet dataset, already made available by the *torchvision* library.

The main assumption we have to make when using transfer learning with a limited amount of data is that the data which the network has been pre-trained on is someway similar to what we are feeding the network with at runtime. Since ImagNet is a huge dataset with about 14M images belonging to 20.000 classes, we can assume it is a very good starting point.

There are different approaches that could be taken when pursuing a transfer learning task, depending on the type and the size of training data you have at runtime. The differences among these approaches are in the part of the network kept *frozen*, that means that its weights are not updated since they are believed to lead to significant results as they are. The image below explains when to implement each of these approaches.

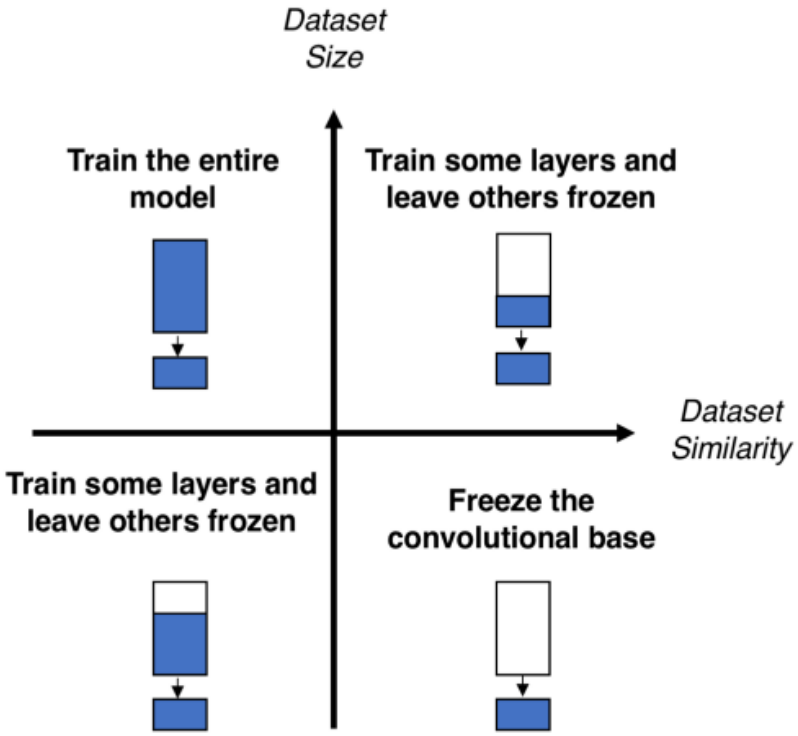


Figure 2 - Transfer Learning approaches

Three different policies have been performed in our analysis: the table shows the different results obtained when re-training the whole network, freezing the convolutional layers and freezing the fully connected layers.

POLICY	FINAL_TRAIN_LOSS	TEST_ACCURACY
train the entire model	0.044	0.82
freeze convolutional layers	0.035	**0.84**
freeze fully connected layers	2.628	0.39

3.1 Train the entire model

First, the whole network is re-trained on the new data. We may be tempted to use the hyperparameters that gave us the best results in the previous step, when training from scratch. Actually, this strategy only makes the loss exploding after a few epochs, since the *LR* is too high: the idea behind transfer learning is to *finetune*, that means to slightly and slowly change the initial weights, thus requiring a quite low learning rate. That's why all the three sets of hyperparameter considered show a lower *LR*.

LR	NUM_EPOCHS	STEP_SIZE	GAMMA	MAX_VAL_ACC
0.001	30	20	0.1	0.82
0.005	20	10	0.1	**0.85**
0.0005	40	30	0.5	0.82

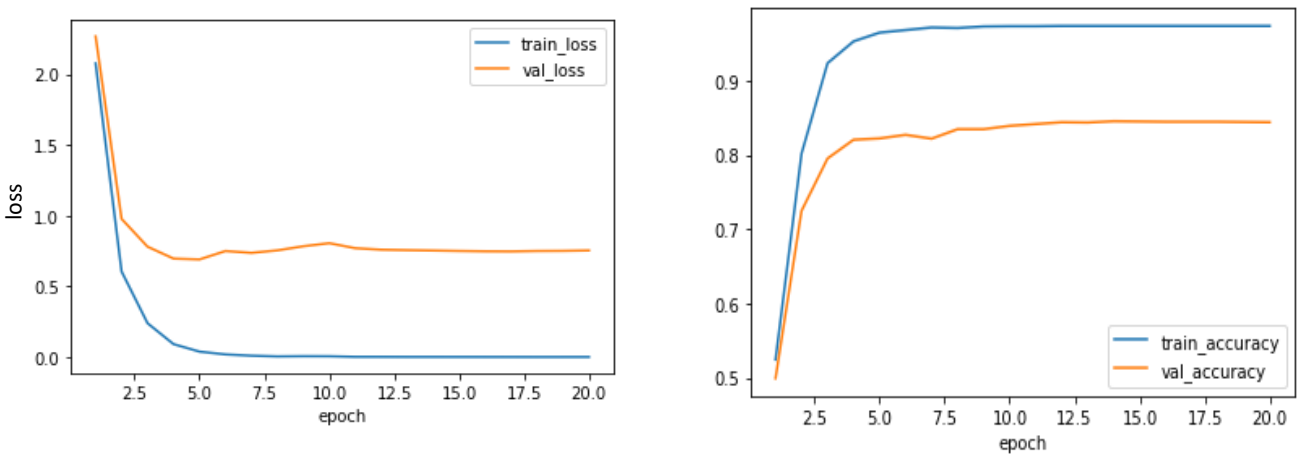


Figure 3 - Loss and accuracy behavior for the best set of hyperparameters

By lowering the LR we are able to obtain way better results in terms of accuracy compared to the training from scratch and to get them after a few epochs, thus confirming the power of the transfer learning. Nevertheless, there is still a form of overfitting, even though it is reduced. For the next steps, when some layers are frozen, only the best set of hyperparameters is used.

3.2 Freeze the convolutional layers

According to the schema shown in *Figure 2*, this would be the best approach, and the results on the test set confirm the theory. By freezing the convolutional layer, we basically trust the feature extraction task performed by the pre-trained network, that makes perfectly sense: even though the classes in ImageNet are different from the ones we are exploring now, the network has learned how to recognize paths common to images in general (like edges, gradients, different shapes, etc..) and to embed this information in the 4096 features vector regardless the class they belong to, and this can be very useful for every image classification task.

The training phase focuses on the fine-tuning of the actual classifier, the FC layers, whose weights need to be problem-dependent.

3.2 Freeze the fully connected layers

According to what is above mentioned about the feature extraction and actual classification process, this policy makes little sense: in fact, the result on the test set is incredibly lower. Moreover, the structure itself of the last layer of the network has been changed to fit our problem (we need 101 output probabilities): we can't expect the weights of a different-structured network to well fit our problem.

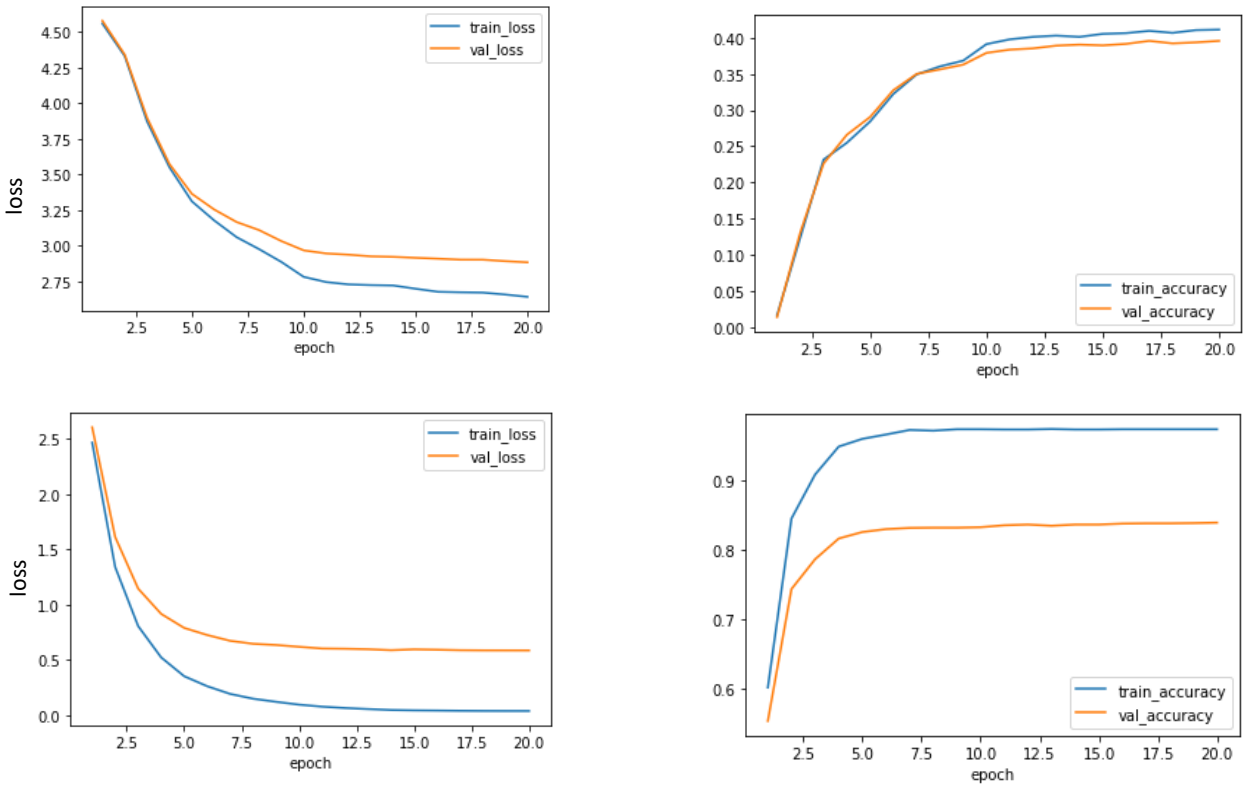


Figure 4 - comparison between freezing different part of the network: FC-layers (above) and conv-layers (below)

4. Data Augmentation

As mentioned before, deep learning basically always benefits from having more amount of data. Unfortunately, the Caltech-101 dataset is quite small. Nevertheless, we can artificially increase the dataset size by applying some transformations to the training images, though preserving their labels. This should help preventing the overfitting we have witnessed so far by adding some variety to the input data, even though the final accuracy may be negatively affected if these variations that we include by transforming the training images are not present in the test set.

The three different sets of transformation pipelines used are:

- *CS-HF*: the image is jittered in **contrast** and **saturation**, then it is **horizontally flipped**;
- *H-RP*: the image is jittered in **hue**, then the **random perspective** transformation is applied;
- *B-GS-R*: the image is jittered in **brightness**, then the relative **grayscale** is taken and finally a **random rotation** is applied;

Each of these pipelines is applied only at training time with a 0.5 probability. In other words, when a batch is taken during the training phase, half of its images are augmented, while the other half are kept in their original state.

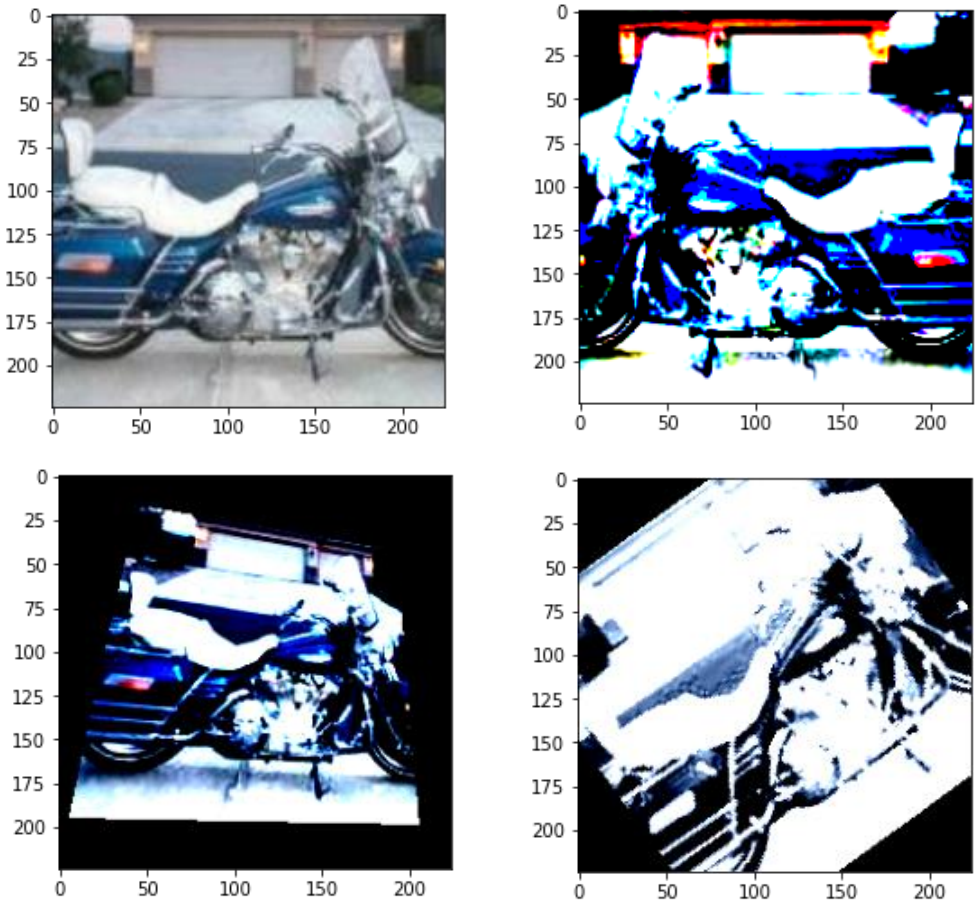


Figure 5 - original (top-left), CS-HF (top-right), H-RP (bottom-left), B-GS-R (bottom-right)

The results, obtained training the whole network using the previously found best set of hyperparameters, are shown in the table below. As a measure to evaluate overfitting, the differences between train and validation accuracies and loss are reported.

AUG_TYPE	TEST_ACCURACY	ACCURACY_GAP	LOSS_GAP
CS-HF	0.85	0.12	0.66
H-RP	0.83	0.15	0.75
B-GS-R	0.81	0.16	0.77

It looks like there isn't a big difference both in terms of accuracy and overfitting between the results obtained with and without data augmentation. Conversely, the more complex the transformation, the lower the accuracy on the test set: this may happen when the dataset is quite homogeneous, and the transformations performed on the training set just contribute to some kind of noise.

5. Extra: other networks

Other networks, more complex than AlexNet, may be used. Here, the results achieved by using such networks are reported, compared to those obtained using AlexNet. All the networks have been initialized with the ImageNet weights and no data augmentation technique has been used.

NET	TEST_ACCURACY	TEST_ACCURACY_FREEZE
AlexNet	0.82	0.84
Resnet18	0.87	0.91
Resnet50	0.91	0.93

More complex networks generally lead to better results: on the other hand, they are also slower to train and require more resources.