Aprendizagem Profunda

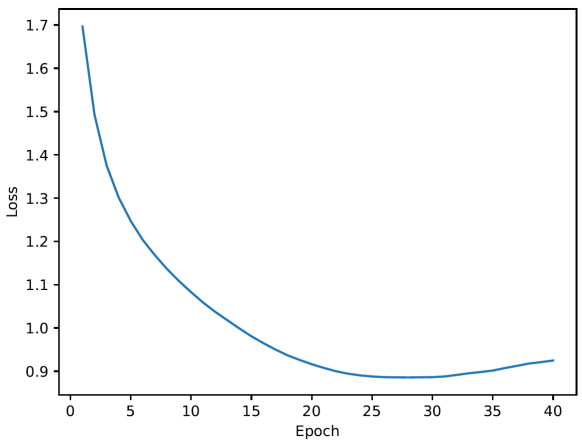
Homework 2

INSTITUTO SUPERIOR TÉCNICO

Ricardo de Jesus Vicente Tavares, 113368

QUESTION 2

1. Convolutional Network



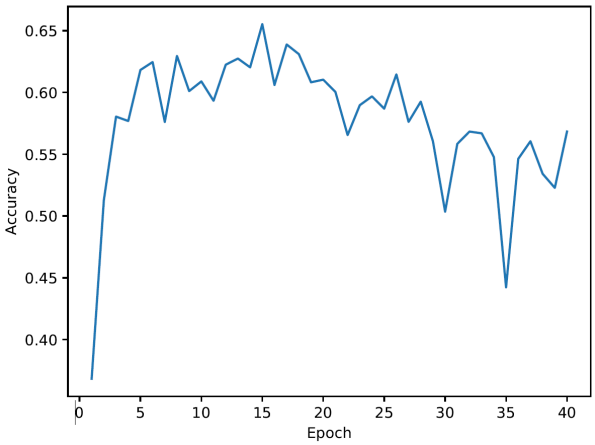
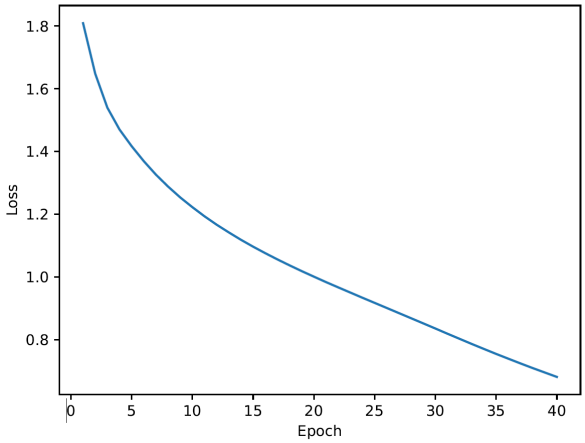


Fig. 1 and 2. Training Loss and Validation Accuracy for Convolutional Network with lr=0.1

```
Valid acc: 0.5684
Final Test acc: 0.5743
```

Fig. 3. Final test and validation accuracy for Convolutional Network with lr=0.1



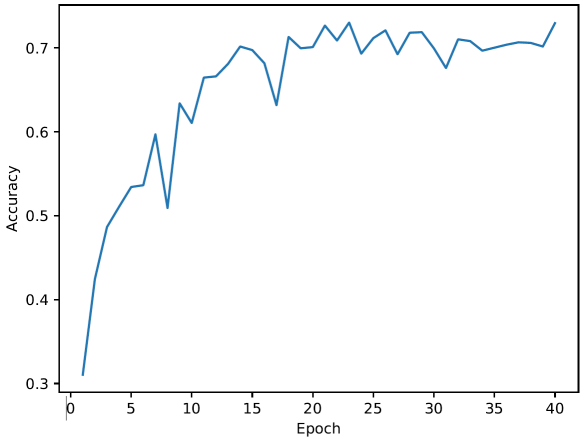


Fig. 4 and 5. Training Loss and Validation Accuracy for Convolutional Network with lr=0.01

```
Valid acc: 0.7293
Final Test acc: 0.6970
```

Fig. 6. Final test and validation accuracy for Convolutional Network with lr=0.01

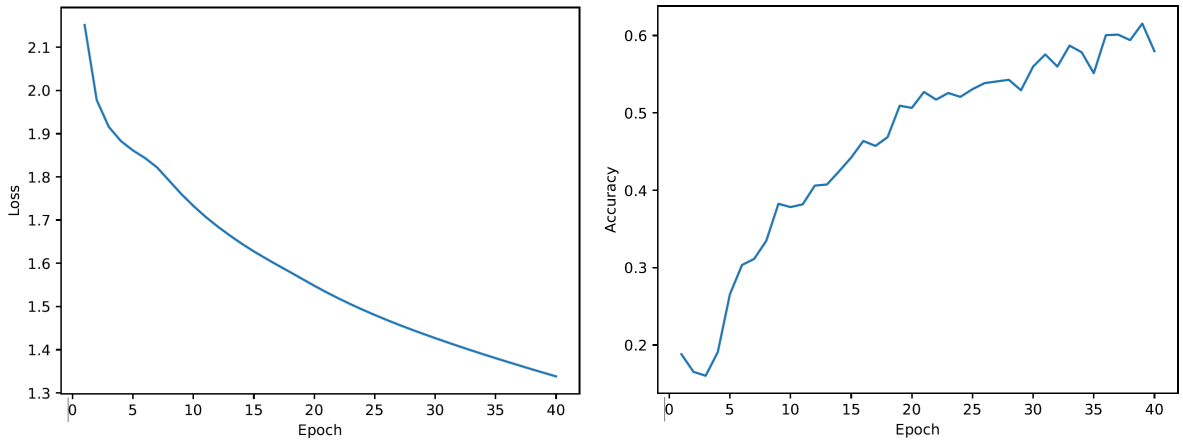


Fig. 7 and 8. Training Loss and Validation Accuracy for Convolutional Network with lr=0.001

```
Valid acc: 0.5798
Final Test acc: 0.5743
```

Fig. 9. Final test and validation accuracy for Convolutional Network with lr=0.001

The learning rate of best configuration is 0.01.

2. Convolutional Network with AdaptiveAvgPool and BatchNorm

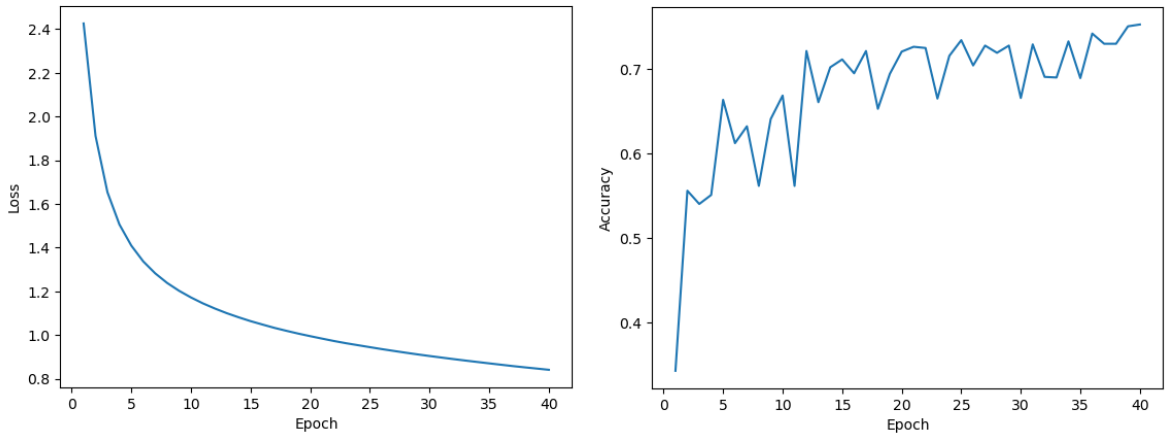


Fig. 10 and 11. Training Loss and Validation Accuracy for Convolutional Network with AdaptativeAvgPool, BatchNorm and lr=0.01

```
Valid acc: 0.7528
Final Test acc: 0.7460
```

Fig. 12. Final test and and validation accuracy for Convolutional Network with AdaptativeAvgPool, BatchNorm and lr=0.01

3. Convolutional Network: number of trainable params

Let's go through the calculations carefully to understand why removing BatchNorm changes the number of trainable parameters from 752,640 to 5,340,160:

1. With BatchNorm
The total number of parameters for the convolution layers is 93,248:

Parameters of conv1: (3×32×3×3)+32 = 896
Parameters of conv2: (32×64×3×3)+64 = 18,496
Parameters of conv3: (64×128×3×3)+128 = 73,856

BatchNorm adds 2 parameters per channel: one for scale (gamma) and one for shift (beta), adding 2 parameters for each channel, 448 in total:

Parameters of conv1: 32×2 = 64 parameters
Parameters of conv2: 64×2 = 128 parameters
Parameters of conv2: 128×2 = 256 parameters

The number of input units for fc1 is simply the number of channels of the last convolution layer: channels[-1] = 128, in our case. The number of parameters for fc1 is:

Parameters of fc1 = (channels[-1]×1×1) × fc1_out_dim + fc1_out_dim
Parameters of fc1 = (128×1×1) × 1024 + 1024 = 132,096

The number of input units for fc2 is obtained by:

Parameters of fc2 = fc1_out_dim×fc2_out_dim + fc2_out_dim
Parameters of fc2 = 524,800

Summing the parameters from all layers, we get the total of 752,640 trainable parameters.

2. Without BatchNorm
Without BatchNorm, the number of parameters increases significantly because the spatial dimensions are preserved. This means the convolution layers don't reduce the output size to 1x1 but instead keep the larger spatial dimensions (e.g., 6x6), which increases the number of inputs to the fully connected layer.

The total number of parameters for the convolution layers is the same as before (93,248).

The number of input units for fc1 is the number of channels of the last convolution layer: channels[-1] = 128, in our case. The number of parameters for fc1 is:

Parameters of fc1 = (channels[-1]×6×6) × fc1_out_dim + fc1_out_dim
Parameters of fc1 = (128×6×6) × 1024 + 1024 = 4,719,616

The number of input units for fc2 is the same as with BatchNorm (524,800).
Summing the parameters from all layers, we get the total of 5,340,160 trainable parameters.

3. Accuracy, training loss and trainable parameters
With 752,640 trainable parameters, the model achieved 74.60% test accuracy, while with 5,340,160 parameters, the accuracy dropped to 69.70%; moreover, in the first case, the decrease in training loss is faster, meaning the model converges more quickly.

It is interesting to note that, in neural network models, the number of parameters is not the only factor that influences accuracy and loss decrease. While it is intuitive that more parameters lead to a more complex model and possibly better performance, in some cases, such as ours, a higher number of parameters can actually reduce accuracy and delay the decrease in loss. There are a few reasons why this might happen:

1. **Overfitting**: More parameters in a model mean a greater ability to learn patterns in the data, but this can also lead to overfitting, where the model becomes too fitted to the training data, capturing not only the relevant features but also the noise in the data. The model may perform very well on the training set but its performance on the test set may be worse because it has learned to memorize specific training details that do not apply to new data.

2. **Insufficient Regularization**: Regularization is essential when the number of parameters is high. If we don't use regularization (such as dropout, L2 regularization, or, in our case, BatchNorm), the model can easily overfit to the training data. In the case of the model with 5,340,160 parameters, we may need, among other things, more regularization (such as stronger dropout, BatchNorm, etc.).

Additionally, more parameters increase the training time due to the higher number of operations and gradient calculations required, making the training process more complex and slower. Furthermore, models with more parameters require more weight updates, demanding more iterations and computational resources.

4. Small kernels in Convolutional Network layers

SMALL KERNELS

In our code, **small kernels** refer to the kernel sizes of the convolutional layers, defined as kernel_size=3 in the ConvBlock instances. These kernels are widely used for the following reasons:

1. **Local Pattern Capture**: Small kernels like 3×3 are effective at capturing local patterns while maintaining computational efficiency.
2. **Parameter Efficiency**: They significantly reduce the number of parameters compared to larger kernels, making the model more efficient. Architectures like VGG and ResNet extensively use 3×3 kernels due to their balance of effectiveness and computational efficiency.
3. **Hierarchical Feature Extraction**: Using multiple small kernels sequentially can mimic the effect of a larger receptive field. For example, two 3×3 kernels stacked together create an effective receptive field equivalent to a 5×5 kernel but with fewer parameters.
4. **Fine-Grained Detail**: Small kernels focus on details, helping the network learn local features like edges and textures, which are crucial for tasks such as image recognition.

The drawback of small kernels is that they have a limited receptive field, requiring multiple layers to capture larger patterns, which increases the network depth and computational cost. This can lead to slower performance, more parameters, and a higher risk of overfitting. Additionally, very deep networks with small kernels may face challenges like the vanishing gradient problem, making training more difficult. To overcome this disadvantage, techniques such as the following can be used:

1. **Dropout and Batch Normalization**: These techniques help prevent overfitting in deep networks, which can occur when increasing the depth of the network to compensate for the limitation of small kernels.
2. **Adaptive Pooling**: Using this technique, which adjusts the pool size based on the input feature map's size, instead of fixed pooling can help reduce dimensionality without losing important information, especially in deeper networks, where the loss of information can be a concern.

POOLING LAYERS

The **pooling layers** are implemented, in our code, using nn.MaxPool2d in the ConvBlock class. Pooling layers are used to downsample the feature maps and bring several benefits:

1. **Dimensionality Reduction**: Pooling decreases the spatial dimensions (width and height) of the feature maps, reducing computational complexity in subsequent layers.
2. **Translation Invariance**: By summarizing local regions (e.g., taking the maximum or average value), pooling makes the network more robust to small translations or distortions in the input.
3. **Focus on Key Features**: Max pooling retains the most prominent feature in each region, enabling the network to focus on critical patterns.
4. **Regularization**: By reducing feature map sizes, pooling discourages overfitting, as the network learns general patterns instead of memorizing specific details.

As drawback, pooling layers can lead to loss of important features and spatial details. This can impact tasks that require precise spatial information. Additionally, pooling does not learn parameters, limiting adaptability. To overcome this disadvantage, we used **AdaptiveAvgPool2d**, which applies average pooling in an adaptive way, meaning it automatically adjusts the output size to the desired number of elements, regardless of the input dimensions. In our code, when we wanted the output to have a specific dimension (1x1), AdaptiveAvgPool2d computes the average of the input values in such a way as to produce the output with the desired size, preserving the global features of the image without the abrupt loss of spatial information that occurs with fixed pooling.

QUESTION 3

1. Validation CER, Test CER and WER

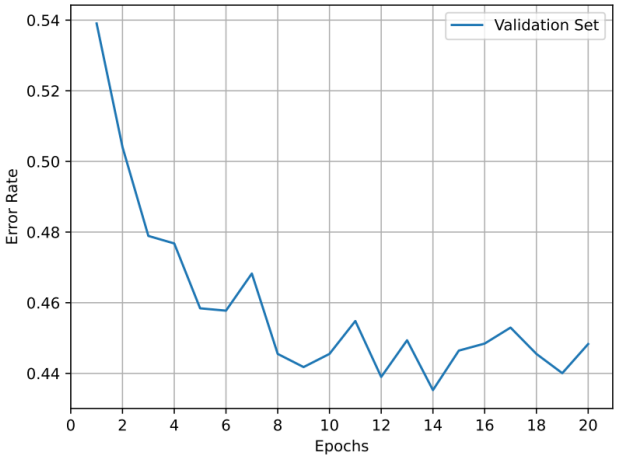


Fig. 13. Validation Character Error Rate (CER)

```
Test CER: 0.4354, Test WER: 0.9040
```

Fig. 14. Test Character Error Rate (CER) and Word Error Rate (WER)

2. Validation CER, Test CER and WER with Bahdanau Attention mechanism

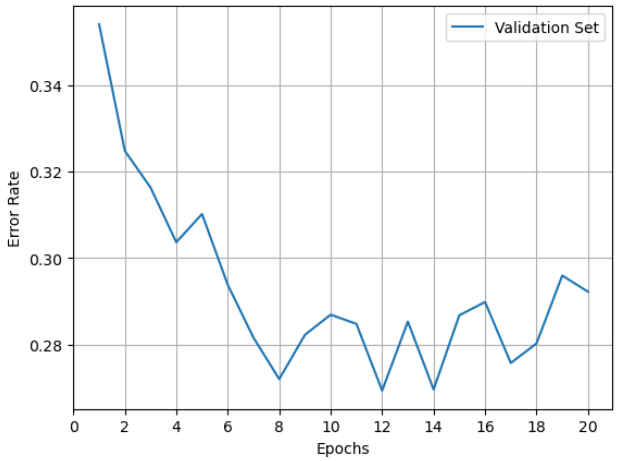


Fig. 14. Validation Character Error Rate (CER) with Bahdanau Attention mechanism

```
Test CER: 0.2677, Test WER: 0.8090
```

Fig. 14. Test Character Error Rate (CER) and Word Error Rate (WER) with Bahdanau Attention mechanism

3. Validation CER, Test CER and WER with Bahdanau Attention mechanism and nucleus sampling

```
Test CER: 0.3139, Test WER: 0.8550
Printing first 10 examples with multiple predictions:
('remainder', {'remander', 'remender'})
('misperception', {'mesperseption', 'misperception'})
('crudités', {'cruditai', 'croudite', 'cruditay'})
('slogger', {'slogger', 'slogar'})
('Jingchuan', {'gingoen', 'ginglwan', 'jinguan'})
('scare', {'scaur', 'skeer', 'skear'})
('glossal', {'glossal', 'glosel', 'glosal'})
('touchedness', {'tutudenis', 'tutuddice', 'tutuddecis'})
('alkaloses', {'alcoloses', 'alcholoses', 'alcolosies'})
('companions', {'companuns', 'companions'})
Test WER@3: 0.7550
```

Fig. 15. Test Character Error Rate (CER), Word Error Rate (WER) amd WER@3 with Bahdanau Attention mechanism and nucleus sampling