Homework 06

IANNWTF 20/21

Submission until 09 Dec 23:59 via https://forms.gle/H31ckx5251Qg3Ndm6

Welcome back to the sixth homework for IANNWTF. This week we have learned about advanced CNN architectures.

In this homework, we want to again build a CNN classifying pictures from the famous Cifar10 dataset. Your goal is to implement simple versions of the two main CNN architectures introduced to you this week, namely ResNet and DenseNet, and see how well they can handle the dataset.

This week we again try to keep the instructions as short as possible, so that you are challenged to try it yourself. Because the best way to learn something is to do it yourself. If you are lost, check the hints for advice or visit us during the Q&A Sessions (their purpose is to support you with any trouble you encounter, so really feel free to drop by!). Also, you can always refer to the code snippets and notebooks provided in courseware.

1 Data set

We will work again with the Tensorflow Cifar10 dataset. https://www.cs.toronto.edu/%7Ekriz/cifar.html. It contains 60.000 coloured images of cells with equal sizes. Each images corresponds to one of 10 categories. The dataset is already implemented as a keras.dataset module and very convenient to load!

Perform necessary or beneficial preprocessing steps.[1] [2]

2 Model

Implement simple versions of the architectures introduced on Courseware.

2.1 ResNet

- Start with implementing a callable **ResidualBlock** class. [3]
  Implement an `__init__` function defining the main building blocks: Your residual

block should consist of multiple alterations of Convolution and Batch Normalization layers. You will need to make sure that the output of the block has the same dimensions as its input. [4]
Implement a `call` function. [5]

- Implement a callable **ResNet** class, consisting of a convolutional layer followed by multiple Residual Blocks and an output layer. [6]
  To further explore your networks behaviour, it might be convenient to implement it in a way that you can easily alter the number of Residual Blocks. [7]
  Implement the networks call function. [8]

2.2 DenseNet

- Start with the Implementation of a callable class for your **Transition Layers**. [9]
- Implement the **Dense Block class**. Remember, a Dense Block is a series of convolutions where the original input is concatenated with the output of the convolution operation. [10] You may want to implement a subclass Block containing the convolution layer [11] and the concatenation implemented in the call function.
  Implement your Dense Block class in a way that you can easily alter the number of Dense Blocks you want to use. [12]
- Implement the **DenseNet** class.
  - You will want to have a set of layers before your Dense Blocks fixing the desired channel dimension you need to pass on to the Dense Blocks.
  - You will want to consecutively alter Dense Blocks and Transition Layers. [13] [14] Remember the use of the growth rate when creating the Transition Layer instances. [15]
  - After the last Dense Block make sure to include a proper set of output layers. [16]

  Implement the respective call function. [17]

3 Training & Analysis

Then train your networks [18]

Explore the relation of the two networks parameters and accuracy on your test data. Also, you maybe would like to compare the number of parameters of the network and how well they perform to the simple convolutional network you trained on the same dataset during last week's homework. [19]

Do not worry about achieving a high test accuracy to pass, rather focus on getting a grasp on the two different architectures. Be aware that training deeper networks like this will take some time and you may want to think of was to speed it up. [20]

4 Outstanding Requirements

General Reminder: Rating yourself as outstanding means that your code could be used as a sample solution for other students. This implies not relying on things not covered in the lecture so far and providing clean understandable code and explanatory comments when necessary.

To receive an outstanding your solution should include the following things:

- A proper Data Pipeline (that clearly and understandably performs necessary pre-processing steps and leaves out unnecessary pre-processing steps).
- Clean understandable implementations of your data set.
- Commments that would help fellow students understand what your code does and why. (There is no exact right way to do this, just try to be empathic and imagine you're explaining topic of this week to someone who doesn't now much about it.)
- Nice visualizations of the losses and accuracies. (You don't have to plot every epoch. Once in the end is enough. Although some print statements to keep track of the performance during training are welcome.)
- Some statements about your findings comparing the different architectures.
- $> 85\%$ accuracy [21]

Notes

[1] Relevant preprocessing steps:

- Shuffling
- Batching, an orientation would be minibatches of size 64. Feel free to also try other batchsizes and see what happens.
- Normalize the images so your network can work with them better. Check Courseware/Image Representation for inspiration.
- One-hot-encode the labels. What should be the depth of the resulting labels?
- Feel free to apply Data Augmentation (shifting, flipping...). Check out Courseware/Regularization/Data Augmentation for that. But be aware, it will slow down your training a lot!

[2] You can simply reuse your data pipeline from last week's homework. If you still have trouble with that, refer to the sample solutions on Courseware or drop by at our Q&A sessions.

[3] As the blocks are basically layers, make sure they are callable by inheriting from tf.keras.layers.Layer/ tf.keras.Model

[4] Good starting point: Use three convolution/batch normalization alterations with kernel sizes of 1,3,1 respectively. Make sure that the number of filters in the last convolutional layer match the channel dimension of your input.

[5] Remember to use a training flag when using Batch Normalization. In the end, the output of your block should be added to the original input of your block.

[6] At best use global average pooling followed by a dense layer (with 10 units) and softmax activation as your readout layers.

[7] A way of doing that is to initialize the model's Residual Blocks in a list whose length is determined by a parameter passed on to the models $\_\_init\_\_$ function. You may want to have a look at inline-lists.

[8] Remember to pass on a training flag that you pass on further to the calls of your residual blocks as they contain Batch Normalization layers.

[9] Use a convolutional layer with kernel size 1 followed by Batch Norm, activation function and a Pooling layer. Check out tf.keras.layers.AveragePooling2D(strides=)

[10] Check out tf.keras.layers.Concatenate()

[11] You will also want to use Batch Norm here.

[12] Again check out inline-lists.

[13] You could implement this by creating an empty list and then by using a for loop alternately append a Dense Block and A Transition Layer to this list.

[14] You will not want to include a Transition Layer after the last Dense Block

[15] Pass on (number of convolution blocks in your dense blocks * growth rate on as the number of channels parameters to your Transition Layer

[16] At best use another set of Batch Norm and global average pooling followed by a dense layer (with 10 units, softmax activation) as your readout layers.

[17] If you used inline list, you will need to loop through them in the call. Again remember to pass on a boolean training flag to all parts where Batch Normalization is used.

[18] Training Hyperparameters for orientation:

- epochs: around 30, depending on your architecture you can train even longer as long as the model is not overfitting
- learning rate: should be small! Try something around 0.001
- optimizer: Adam
- loss: CategoricalCrossEntropy. Check `tf.keras.losses.CategoricalCrossentropy()`
- accuracy: how many items in prediction and target are the same (in the batched vectors)? → take the mean of it

[19]Recall the use of model.summary()

[20]If you are using colab/ have an nvidia GPU make sure you are using it properly. Also recall the use of the decorator @tf.function

[21]We achieved around 60% accuracy after about ten epochs with just around 12000 network parameters. Scaling it up will take some time to train, but if you do it right you should get more than 90% accuracy.