TP4 du cours ROB313 (CNN with CIFAR10)

Auteurs du CR

Zhi Zhou, zhi.zhou@ensta-paris.fr Simon Queyrut, simon.queyrut@ensta-paris.fr

@zroykhi, @queyrusi (repo source)

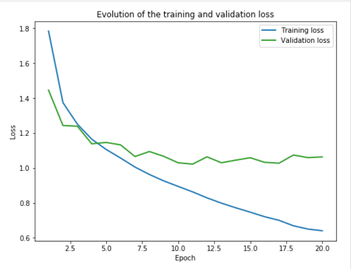
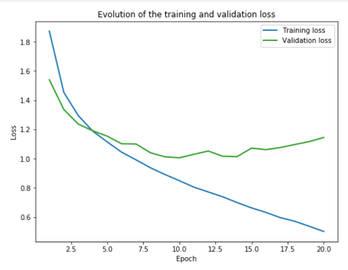
Parameter tuning

With default model below, we test and discuss some important factors of the neural network.

```
class MyConvolutionalNetwork(nn.Module):
    def __init__(self):
        super(MyConvolutionalNetwork, self).__init__()

        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

        #### START CODE: ADD NEW LAYERS ####
        # (do not forget to update `flattened_size`:
        # the input size of the first fully connected layer self.fc1)
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 16, kernel_size=3, stride=1, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.dropout1 = nn.Dropout(p=0.35)

        # Size of the output of the last convolution:
        self.flattened_size = 16 * 8 * 8
        ### END CODE ###

        self.fc1 = nn.Linear(self.flattened_size, 64)
        self.fc2 = nn.Linear(64, 10)

    def forward(self, x):
        """
        Forward pass,
        x shape is (batch_size, 3, 32, 32)
        (color channel first)
        in the comments, we omit the batch_size in the shape
        """
        # shape : 3x32x32 -> 18x32x32
        x = F.relu(self.conv1(x))
        # 18x32x32 -> 18x16x16
        x = self.pool(x)

        #### START CODE: USE YOUR NEW LAYERS HERE ####
        x = F.relu(self.conv2(self.bn1(x)))
        x = self.dropout1(x)
        x = self.pool1(x)

        #### END CODE ####
        ...
```

Dropout layer

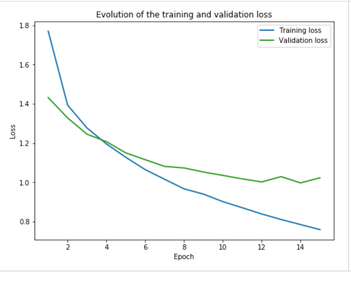
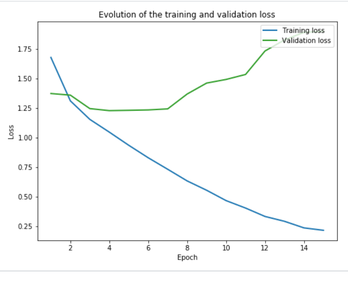
Dropout is a technique used to prevent a model from overfitting. Dropout works by randomly setting the outgoing edges of hidden units (neurons that make up hidden layers) to 0 at each update of the training phase. It offers a very computationally cheap and remarkably effective regularization method.

with dropout layer	without dropout layer
	

As shown in the above images, we train the model twice with the only difference of with or without dropout layer. Randomly turning off units will rise the validation loss sometimes during the training process, which is normal. That's why there are more ups and downs on the validation loss when dropout layer is applied to the model, but overall, the loss is **descending**. On the contrary, without dropout layer, the validation loss keep going high at the end of training, which implies the model is over-fitting.

Pooling layer

A limitation of the feature map output of convolutional layers is that they record the precise position of features in the input. This means that small movements in the position of the feature in the input image will result in a different feature map. Pooling layers provide an approach to down sampling feature maps by summarizing the presence of features in patches of the feature map. This is where a lower resolution version of an input signal is created that still contains the large or important structural elements, without the fine detail that may not be as useful to the task.

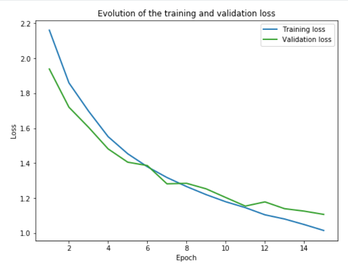
with pooling layer	without pooling layer
	

Obviously, we can see from the above images that without pooling layers, the model is over-fitted. According to previous explaination, this may because the model will be too sensitive with feature map without pooling layers. It proves the importance of pooling layers

Batch normalization

Batch noremalization is the techinique that we normalize the input layer by adjusting and scaling the activations. There are many beneifts by doing so

1. Batch normalization allows each layer of a network to learn by itself a little bit more independently of other layers.
2. Reduces the dependence of gradients on the scale of the parameters or their initial values.
3. Regularizes the model and reduces the need for dropout, photometric distortions, local response normalization and other regularization techniques.
4. Allows use of saturating nonlinearities and higher learning rates.

with batch normalization	without batch normalization
	

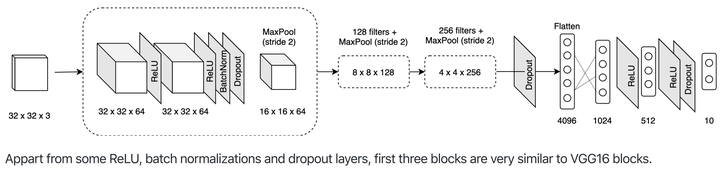
	with batch normalization	without batch normalization
train loss	0.78	1.07
validation loss	1.03	1.11

With batch normalization, the network could achieve better performance.

Optimizor

Just for this section, optimizors will be discussed with regards to a truncated version of VGG16. A complete and adaptated version of VGG16 is tested at the end of the report.

Architecture is shown below:



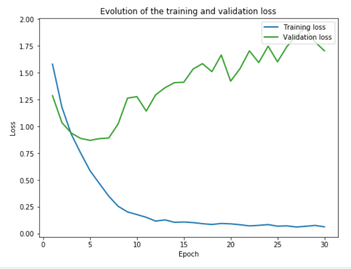
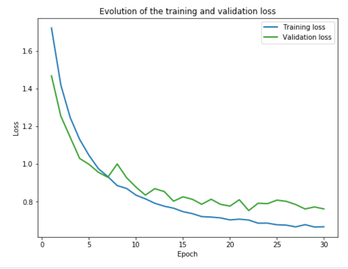
Appart from some ReLU, batch normalizations and dropout layers, first three blocks are very similar to VGG16 blocks.

We use Adam optimizer for starters: a parameter that can be tuned is weight decay. When training NNs, if we set a value to `weight_decay` parameter inside `optim.Adam()`, the weights are multiplied by a factor slightly less than 1. This can be seen as gradient descent on a quadratic regularization term λ.

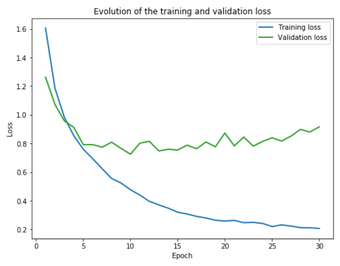
Pytorch handles weight updates as this:

$$W^{(i+1)} = W^{(i)} - \eta G^{(i)}$$
$$G^{(i)} = \frac{dL}{dW^{(i)}} + \lambda W^{(i)}$$

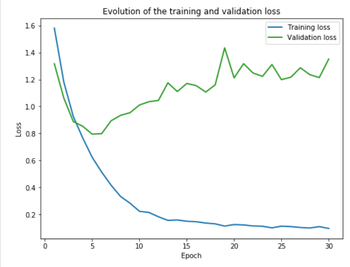
We run several experiments with different λ values:

Truncated VGG16 on Adam optimizor with `weight_decay=0.0`	Truncated VGG16 on Adam optimizor with `weight_decay=0.01`
	
Truncated VGG16 on Adam optimizor with `weight_decay=0.001`	**Truncated VGG16 on Adam optimizor with `weight_decay=0.0001`**

Truncated VGG16 on Adam optimizor with `weight_decay=0.001`



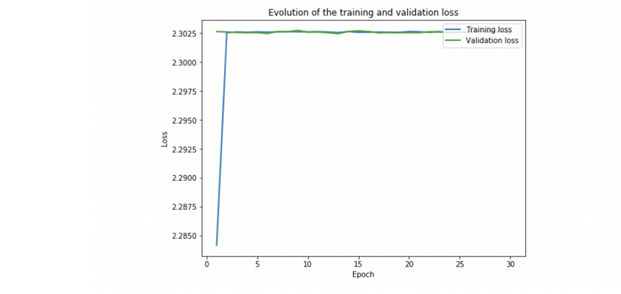
Truncated VGG16 on Adam optimizor with `weight_decay=0.0001`



Performance are as follow (given by `compute_accuracy`):

weight decay factor	train	validation	test
0.0	77.53	71.62	66.30
0.01	73.81	73.66	67.64
0.001	82.19	75.96	70.86
0.0001	79.26	72.98	68.50

We also tested $\lambda = 1$ but results are ridiculous:



Overall, we observe that with smaller weight decay comes weaker confidence in training performance since training performance, albeit very good, fail to validate by a large margin (e.g. $82.19 - 75.96 = 6.23$ when `weight_decay` equals 0.001). Poorly chosen λ can even lead to overfitting when it should actively figth against it.

For most test in this report we therefore use a λ of 0.01 to insure training confidence.

Good performance with this "truncated" model hints at using the full model but with some modification since MaxPooling decreases feature numbers by half every time and that could be dangerous on 32 by 32 pictures.

Number of epochs

The number of epochs also influences the training results. Too few epochs may results in under-fittng whereas too many epochs may lead to over-fitting, which makes it really important to choose a proper number of epochs.

The table below shows the trend of training loss, validation loss and train time with respect to number of training epochs.

num of epochs	10	15	20	25	30
train loss	0.93	0.79	0.67	0.57	0.51
validation loss	1.03	1.01	1.06	1.11	1.17
train time/s	61.31	92.63	114.30	153.08	182.98

The training loss keeps going down with augment of number of epochs, however, the validation loss reachs minimum value at around 15th epochs and then increases. We could early stop the training process at around 15th epochs in order to achieve a better validation result.

Size of mini-batch

Mini-batch gradient descent is a variation of the gradient descent algorithm that splits the training dataset into small batches that are used to calculate model error and update model coefficients. Batch size is a slider on the learning process. A larger mini-batch will provide a stable enough estimate of what the gradient of the full dataset would be but it will be memory-consumed. Smaller size of mini-batch allows the model update more frequently but may converge more slowly.

Below are some other factors may improve the model performance. We did't discuss them due to time limit, but it is worth digging deeper into them

1. number of filters in convolutional layer
2. kernel(filter) size

VGG16

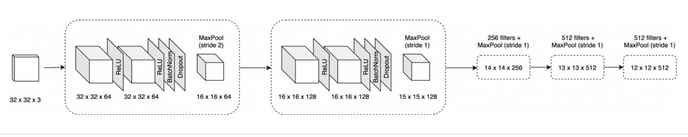
Original VGG16 with `MaxPool` stride 2

Following architecure is a slight adaptation of **VGG16**. (code in notebook)

```
MyConvolutionalNetwork(
  (conv_layer): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): Dropout2d(p=0.05, inplace=False)
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (7): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): ReLU(inplace=True)
    (11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (12): MaxPool2d(kernel_size=2, stride=1, padding=0, dilation=1, ceil_mode=False)
    (13): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (14): ReLU(inplace=True)
    (15): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (16): ReLU(inplace=True)
    (17): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (18): MaxPool2d(kernel_size=2, stride=1, padding=0, dilation=1, ceil_mode=False)
    (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (24): ReLU(inplace=True)
    (25): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (26): MaxPool2d(kernel_size=2, stride=1, padding=0, dilation=1, ceil_mode=False)
    (27): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (28): ReLU(inplace=True)
    (29): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (30): ReLU(inplace=True)
    (31): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (32): ReLU(inplace=True)
    (33): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (34): MaxPool2d(kernel_size=2, stride=1, padding=0, dilation=1, ceil_mode=False)
  )
  (fc_layer): Sequential(
    (0): Dropout(p=0.1, inplace=False)
    (1): Linear(in_features=73728, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Linear(in_features=4096, out_features=10, bias=True)
  )
)
```

Below is a representation of the "big" receptive field part of our experimental CNN (which corrsponds to the five colorful blocks above, which is also `conv_layer` in the snippet above). Dense layers are practically the same so they were not represented:

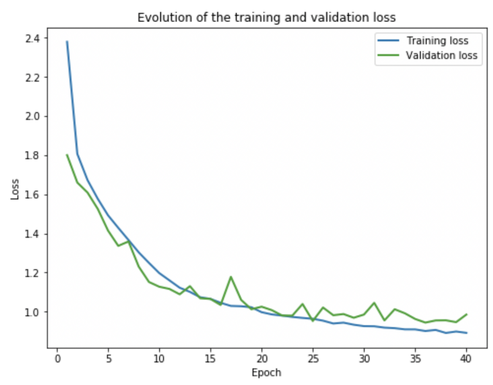
Custom-made VGG16 with 4 blocks of `MaxPool` stride 2



Main differences from original VGG16 are

- Input is of course $(length, width, channels) = (32, 32, 3)$ instead of originally $(224, 224, 3)$
- Except in first block, `MaxPool` was done with a stride 1 instead of 2 originally. Given size of input, it was obvious we wanted to avoid $(1, 1, 512)$ situations at the 5th block.
- We added some ReLU and batch normalizations between the convolutional layers.
- Dropouts were added in every block.
- Output is of size 10 instead of 1000 as displayed in the picture.

complete VGG16 with `MaxPool` stride 1



Experiment was run with Adam optimizor, `weight_decay=0.01`.

Interestingly enough, this adaptation of VGG16 to smaller pictures seems safe from overfitting under 40 epochs even though it has significantly more parameters than previous models. Performances are slightly worse than truncated version of VGG16, though the $|L_{train} - L_{validation}|$ distance looks better off for it.

This model took about one hour to run on colab (with GPU settings on).