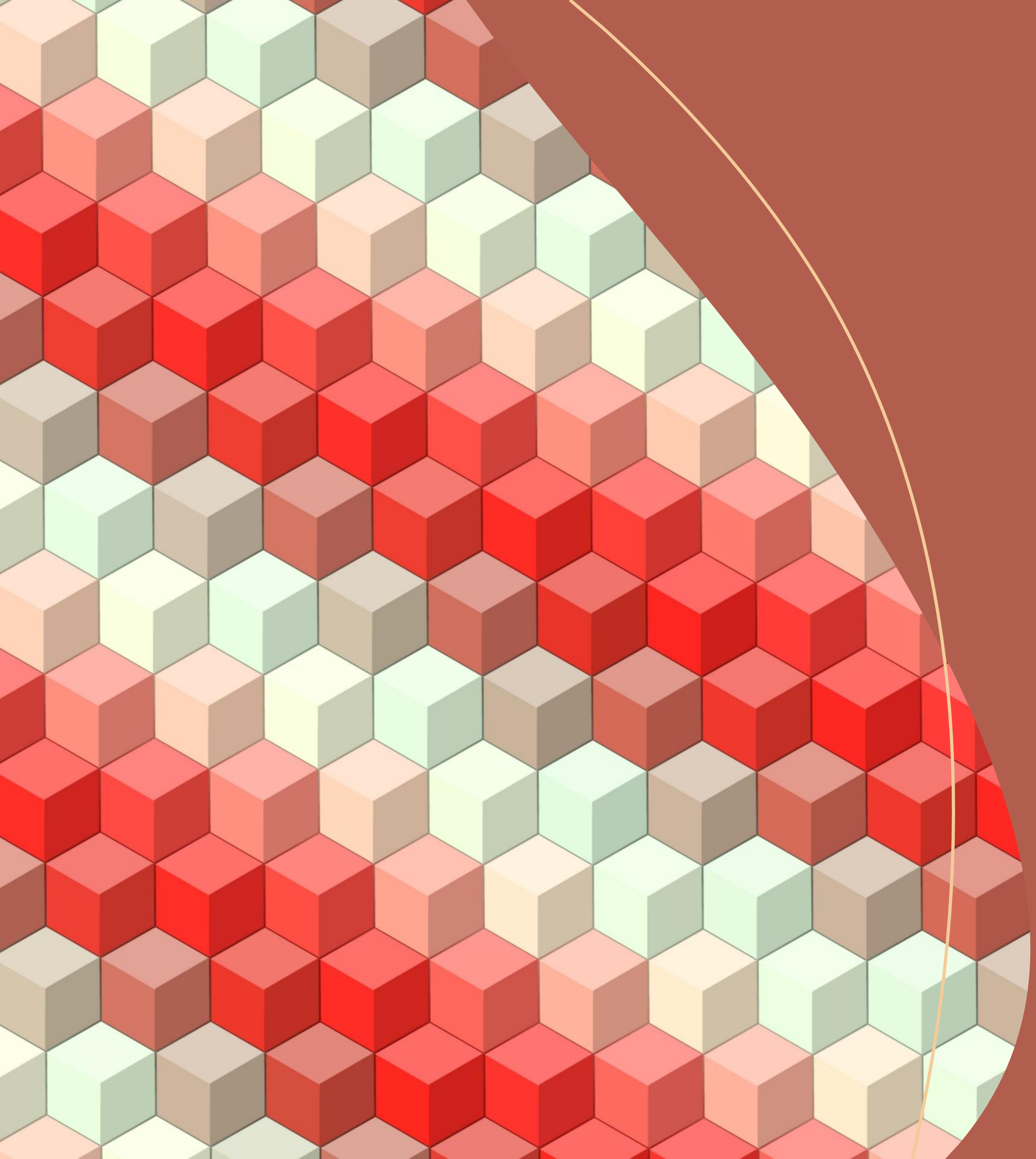


Guide to Implementing Tensor Regression Layers & Tensor dropout into Neural Networks

By: Leonel Ramirez

Prior to Starting

- The following presentation/ tutorial was inspired by Jean Kossafi's work on Tensors
- I would also like to thank Jean Kossafi for answering my questions whenever I had any in reference to Tensorly and Tensorly-Torch
- Most of the material here builds off Jean Kossafi's GitHub which I have linked here
  - It is full of tensor methods and tutorials if you would like to take a deeper dive
- I am simply doing a more hands on approach for beginners looking to implement TRLs and Tensor Dropout into their neural networks for robust performance

Traditional Approaches

(From Jean Kossafi slides, [here])

- Data => CONV => RELU => POOL => Activation tensor
- Flattening looses spatial information
- Can we leverage directly the activation tensor before the flattening?
- Potential space savings
- Performance improvement

Deep Neural Network Architectures

(From Jean Kossafi slides, here)

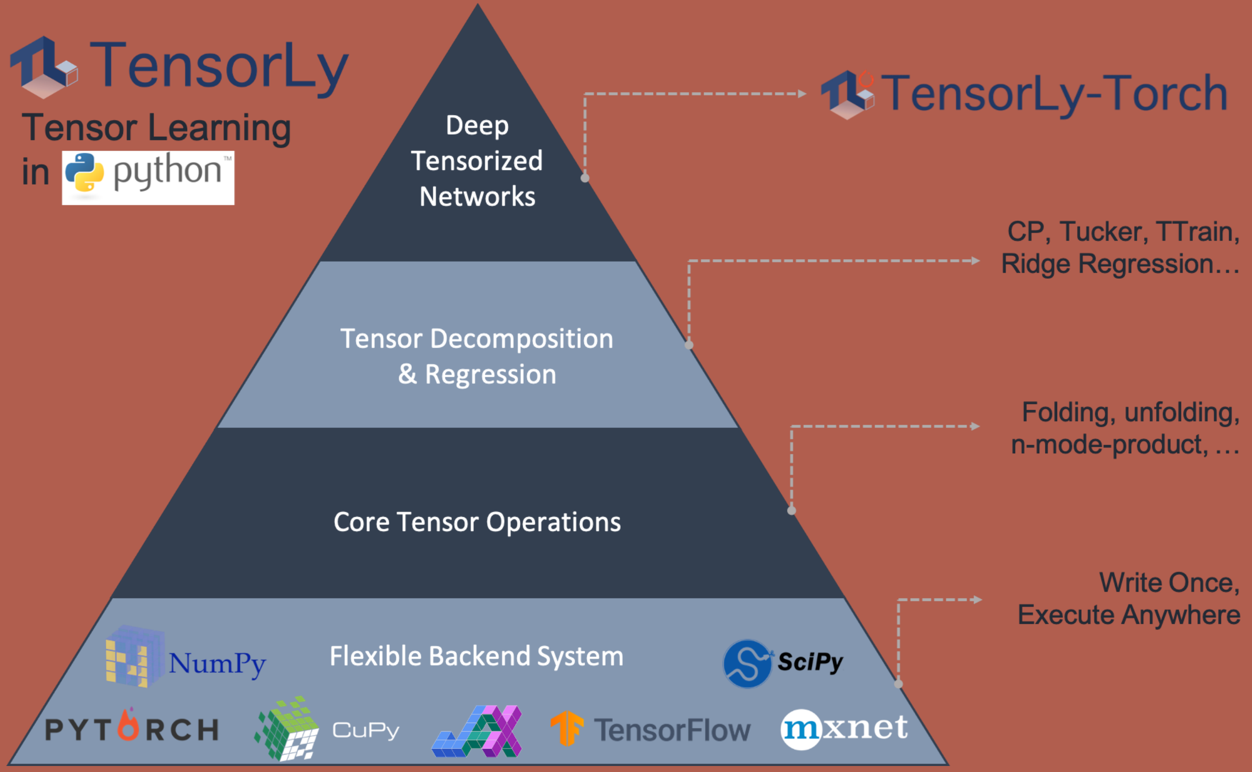
	Total params	Params in FC	%params in FC
AlexNet	61,100,840	58,631,144	96%
VGG-19	143,667,240	123,642,856	86%
ResNet-50	25,557,032	2,049,000	8%
RestNet-101	44,549,160	2,049,000	4.6%

Why Tensor Regression Layers?

- Tensor Regression Networks propose to replace fully connected layers entirely with a tensor regression layer (TRL), which reduces the number of parameters needed within a network.
- This preserves the structure of the multidimensional data (input) by expressing an output tensor as the result of a tensor contraction between the input tensor and some low rank regression weight tensor

What you need to replace FCLs with a TRL

- Packages needed: Pytorch, Tensorly, and Tensorly Torch
  - pip install packages
- An example will be done for a pretrained model and a CNN model from scratch



Implementing TRL and Tensor Dropout into a pretrained model

- To relace the forward method of a predefined torchvision model with your own customized forward function you must first create a custom class using the nn.Module class as its parent
- The example shows the creation of a **CP_TRL_Layer** class which holds the newly created **CP_TRL layer** as well as **Tensor Dropout** being applied to that same factorized layer.
- It is then passed to the existing forward method of the pretrained model.

```
class CP_TRL_Layer(nn.Module):

  def __init__(self, num_classes, rank):

      super(CP_TRL_Layer, self).__init__()

      self.trl = CP_TRL(input_shape = (512, 7, 7), rank = rank,
                        output_shape = num_classes)

      self.trl = cp_dropout(self.trl, p = 0.6)

  def forward(self, x):

    x = self.trl(x)

    return x
```

Implementing TRL and Tensor Dropout into a pretrained model

```
class CP_TRL_Layer(nn.Module):
  def __init__(self, num_classes, rank):
      super(CP_TRL_Layer, self).__init__()
      self.trl = CP_TRL(input_shape = (512, 7, 7), rank = rank,
                             output_shape = num_classes)
      self.trl = cp_dropout(self.trl, p = 0.6)
  def forward(self, x):
     x = self.trl(x)
     return x
```

```
from torchsummary import summary
# summary(model_name, (number_of_filter, pixel_size, pixel_size))
summary(model, (3, 224, 224))
```

- The **input_shape** is determined from the last output of the previous layer and the number of filters. It is worth noting that image size effects the last output.
  - The shape can be seen using the **summary** function
- The **rank** of a tensor is the number of indices required to uniquely select each element of the tensor.
  - No intuition/formula for ideal rank, one must find the rank best suited for their application
- The **output_shape** is the number of classes you have for classification
  - For example, facial emotion recognition (FER2013) dataset contains 7 different emotions, so simply plug 7 into **num_classes**
- The **p** is simply the probability of dropping and is calculated as followed:
  - **p** = 1 - **p**

Implementing TRL and Tensor Dropout into a pretrained model

- Once the pretrained **vgg16** is assigned a name, in this case **model,** its attributes can now be accessed with the dot operator.
- The **.classifier** attribute in this case had 7 layers in its classifier block which needed to be removed since it contained the fully connected layers of the vgg16.
  - The result of this is an empty classifier block where information/ data will just pass through
  - Can be seen by simply calling the **model** (seem to the right)
- To implement the TRL w/ Tensor Dropout, the **.avgpool** attribute must be accessed so that layer can be replaced with the **CP_TRL_Layer** class, created previously.
  - Changes be seen by **summary**(model, (channels, image_size, image_size)) or simply calling the **model**

```
device = 'cpu’

#device = 'cuda:0'

model = models.vgg16(pretrained = True)

for param in model.parameters():

  param.requires_grad = False

model.classifier = nn.Sequential(*list(model.classifier.children())[:-7])

model.avgpool = CP_TRL_Layer(num_classes = 10, rank = 10000, p = 0.4)

model.to(device)

summary(model, (3, 224, 224))

model
```

```
(avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
(classifier): Sequential(
  (0): Linear(in_features=25088, out_features=4096, bias=True)
  (1): ReLU(inplace=True)
  (2): Dropout(p=0.0, inplace=False)
  (3): Linear(in_features=4096, out_features=1000, bias=True)
  (4): ReLU(inplace=True)
  (5): Dropout(p=0.0, inplace=False)
  (6): Linear(in_features=1000, out_features=10, bias=True)
)
```

```
(avgpool): CP_TRL_Layer(
  (trl): CP_TRL(
    (factors): ParameterList(
        (0): Parameter containing: [torch.FloatTensor of size 512x10000]
        (1): Parameter containing: [torch.FloatTensor of size 7x10000]
        (2): Parameter containing: [torch.FloatTensor of size 7x10000]
        (3): Parameter containing: [torch.FloatTensor of size 10x10000]
    )
  )
)
(classifier): Sequential()
```

Implementing TRL into a CNN model

```
class ConvNeuralNet(nn.Module):

    def __init__(self, num_classes):

        super(ConvNeuralNet, self).__init__()

        self.conv_layer1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3)

        self.max_pool1 = nn.MaxPool2d(kernel_size = 2, stride = 2)

        self.conv_layer2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3)

        #self.fc1 = nn.Linear(64*13*13, 128) # 128 is a random choice by me could be anything

        #self.relu1 = nn.ReLU()

        #self.fc2 = nn.Linear(128, num_classes)

        self.trl = CP_TRL(input_shape=(64, 13, 13), output_shape=(7), rank=100)
```

The current architectures main difference from the previous one would be that the Flatten layer and all fully-connected layers have been replaced by a Tensor Regression Layer

```
# Progresses data across layers

    def forward(self, x):

        out = self.conv_layer1(x)

        out = self.max_pool1(out)

        out = self.conv_layer2(out)

        #out = out.reshape(out.size(0), -1)

        #out = self.fc1(out)

        #out = self.relu1(out)

        #out = self.fc2(out)

        out = self.trl(out)

        return out
```

Output shapes be seen by **summary**(model, (pic_channels, image_size, image_size))

```
model = ConvNeuralNet(num_classes)

from torchsummary import summary

summary(model, ((3,32,32)))
```

Layer (type)	Output Shape
Conv2d-1	[-1, 32, 30, 30]
MaxPool2d-2	[-1, 32, 15, 15]
Conv2d-3	[-1, 64, 13, 13]
CP_TRL-4	[-1, 7]

Sidenote: the output of a layer is the input to the next

Sources

- https://github.com/JeanKossaifi/caltech-tutorial/blob/master/slides/5-tensor%2Bdeep.pdf
- http://tensorly.org/torch/stable/modules/api.html#tensor-regression-layers
- https://discuss.pytorch.org/t/how-can-i-replace-the-forward-method-of-a-predefined-torchvision-model-with-my-customized-forward-function/54224/70
- https://github.com/JeanKossaifi