From Knowledge Distillation to BAN:
A practical test on real world datasets

Federico Alfano

October 12, 2020

1 Introduction

In this report we are going to have a look in depth about the Knowledge Distillation. Some papers were written about that, in particular, the idea was to have a student who distills knowledge from the teacher and who has comparable reliability but higher speed, so that, unlike the latter can be used in a production environment.
In the paper[1] that we are going to deepen, a further step is taken, the distillation of knowledge is done by a student of same complexity to that of the teacher. The authors in this case observe an improvement in the performance of the student networks, and surprisingly the improvement also occurs with subsequent generations of students. In the paper the new networks are called Born Again Networks. My work will be to replicate those results in a practical way with real datasets, trying to explain as best as possible every step, making available all the code produced during my experiments. I'll also try to experiment some variant just to annotate what are the changes.
In the introduction I will just prepare the environment that will be used for the next experiments, while in the following parts of the report I will implement the algorithm and then perform all the tests.

1.1 Residual Network

One of the models used in the reference paper Wide Residual Network, presented in another paper indicated in the bibliography[5]. In essence, the creators of the new model (WRNs) suggest that a less "deep" structure can minimize the problem of diminishing feature reuse so that even a fraction of improvement needs to double the layers. Until now, neural networks were made deeper and deeper to reduce the number of parameters, but the authors of the paper found that compared to deep resnet presented here [7] they needed 50 times less layers. They claim that a WResNet with 16 layers has an accuracy comparable to a DResNet with 1000 layers, and is also faster in training.
Looking at the code provided by the authors, there is only one implementation with the pytorch framework, but the intention is to complete the task with Tensorflow 2.0, so it is interesting to have a version of it also for the framework I used in this work.

The first step will be the coding of the model, which I think it may be interesting to develop also using the Tensorflow Subclassing API in order to have a clean and usable version later, even at the cost of meeting some problems that will be mentioned later.
From the paper[5] we can see the different kind of blocks

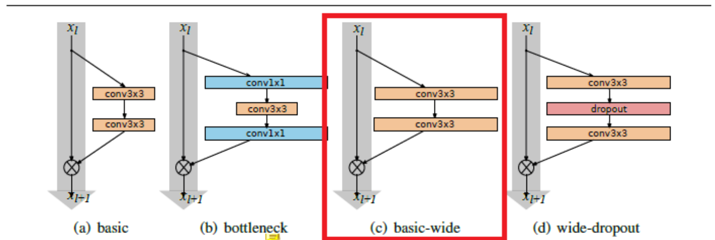

Figure 1: Various residual blocks used in the paper. Batch normalization and ReLU precede each convolution (omitted for clarity)

Let's see the code of the Residual Layer

Listing 1: The code of the Residual Layer

```
class ResidualBlock(layers.Layer):
  def __init__(self, filters, kernel_size, dropout,
               dropout_percentage, strides=1, **kwargs):
    super(ResidualBlock, self).__init__(**kwargs)
    self.conv_1 = layers.Conv2D(filters, (1, 1), strides=strides)
    self.bn_1 = layers.BatchNormalization()
    self.rel_1 = layers.ReLU()
    self.conv_2 = layers.Conv2D(filters, kernel_size, padding="same",
                                strides=strides)
    self.dropout_layer = layers.Dropout(dropout_percentage)
    self.bn_2 = layers.BatchNormalization()
    self.rel_2 = layers.ReLU()
    self.conv_3 = layers.Conv2D(filters, kernel_size, padding="same")
    self.add = layers.Add()
    self.dropout = dropout
    self.strides = strides


  def call(self, inputs, training=None):
    x = inputs
    if self.strides > 1:
      x = self.conv_1(x)
    res_x = self.bn_1(inputs)
    res_x = self.rel_1(res_x)
    res_x = self.conv_2(res_x)
    if self.dropout:
      res_x = self.dropout_layer(res_x, training=training)
    res_x = self.bn_2(res_x)
    res_x = self.rel_2(res_x)
    res_x = self.conv_3(res_x)
    inputs = self.add([x, res_x])
    return inputs
```

The block is the one shown in the figure, with a parameter to activate the Dropout. The get_config method instead is simply used to allow the user to save the model. In the paper there is also the "bottleneck layer" that is neglected, and I will do the same in my work, focusing on the "basic" type.
The model instead is an aggregation of ResidualLayers that takes in input 2 parameters: k and d (d in the paper is called "l") , which are respectively the widening factor and the deepening factor.
Let's see the code of the Model:

Listing 2: The code of the Wide Residual Network

```
class WideResidualNetwork(models.Model):
  def __init__(self, n_classes, d, k, kernel_size=(3, 3),
               dropout=False, dropout_percentage=0.3, strides=1, includeTop=True, **kwargs):
    super(WideResidualNetwork, self).__init__(**kwargs)
    if (d-4)%6 != 0:
      raise ValueError('Please choose a correct depth!')

    self.dropout = dropout
    self.dropout_percentage = dropout_percentage
    self.N = int((d - 4) / 6)
    self.k = k
    self.d = d
    self.includeTop = includeTop
    self.kernel_size = kernel_size

    self.bn_1 = layers.BatchNormalization()
    self.rel_1 = layers.ReLU()
    self.conv_1 = layers.Conv2D(16, (3, 3), padding='same')
    self.conv_2 = layers.Conv2D(16*k, (1, 1))
    self.dense = layers.Dense(n_classes)

    self.res_block_1 = [ResidualBlock(16*self.k, self.kernel_size, self.dropout,
     self.dropout_percentage) for _ in range(self.N)]
    self.res_single_1 = ResidualBlock(32*self.k, self.kernel_size, self.dropout,
     self.dropout_percentage, strides=2)
    self.res_block_2 = [ResidualBlock(32*self.k, self.kernel_size, self.dropout,
     self.dropout_percentage) for _ in range(self.N-1)]
    self.res_single_2 = ResidualBlock(64*self.k, self.kernel_size, self.dropout,
     self.dropout_percentage, strides=2)
    self.res_block_3 = [ResidualBlock(64*self.k, self.kernel_size, self.dropout,
     self.dropout_percentage) for _ in range(self.N-1)]
    self.pooling = layers.GlobalAveragePooling2D()
    self.activation_layer = layers.Activation("softmax")



  def call(self, inputs, training=None):
    x = self.bn_1(inputs)
    x = self.rel_1(x)
    x = self.conv_1(x)
    x = self.conv_2(x)
    for layer in self.res_block_1:
      x = layer(x, training=training)

    x = self.res_single_1(x, training=training)
```

```
for layer in self.res_block_2:
  x  = layer(x, training=training)

x = self.res_single_2(x, training=training)

for layer in self.res_block_3:
  x = x = layer(x, training=training)

x = self.pooling(x)
x = self.dense(x)
if self.includeTop:
    x = self.activation_layer(x)

return x
```

In both cases I omitted the *get_config* function, as it is not useful for understanding the model.

2 The Algorithm

2.1 Knowledge Distillation

All the paper work is based on knowledge distillation, which is a technique that allows a neural network to distil knowledge from a master neural network, without knowing anything about the dataset.
This process has been studied previously in some papers[2][3][4], but with the sole purpose of building a lighter and simpler student model that could be used in a production environment, starting from a much more complex master, but also keeping good performances.
In the reference paper the idea is to use the same technique but this time applying it to the same model or to a model of comparable complexity. What we saw is that often the student model was superior to the master. All this seems to be the result of a more complete information contained in the output of the master model, in fact it does not only provide a negative or positive value for each class of reference, but also a probability distribution that makes the model aware of how two classes are "close" and "confusable".
There are different ways to work with knowledge distillation, that is why I decided to have an implementation as generic as possible. So I coded a method that would take in input a dataset, a teacher and a student. As a first step the dataset is encapsulated in a custom generator that iterating returns the as X, the same as the dataset, and as y, instead of ground truth, the prediction of the master.
But now let's see the code:

Listing 3: The code for distilling knowledge

```
def distill_knowledge(teacher_model, student_model, train_dataset, valid_data,
                      student_callbacks, epochs, batch_size, steps_per_epoch):
  def custom_generator(train_dataset, t_model):
    for (x, y) in train_dataset:
      y_targets = teacher_model(x)
      yield (x, y_targets)

  s_history = student_model.fit(custom_generator(train_dataset,
                                                 t_model=teacher_model),
                      epochs=epochs,
                      steps_per_epoch=steps_per_epoch,
                      callbacks = student_callbacks,
                      validation_data=valid_data)

  return s_history
```

2.2 Born Again Network

The main algorithm presented in the paper[1] trains the teacher until convergence, after which, it initializes a student and uses the output of the master's softmax as the student's target. The process is repeated for several times, and it is observed that at some point there will be no improvement.
The study also indicates the possibility of improving performaces using an ensemble formed by the various generations of students. The output will therefore be an average of the students' output.

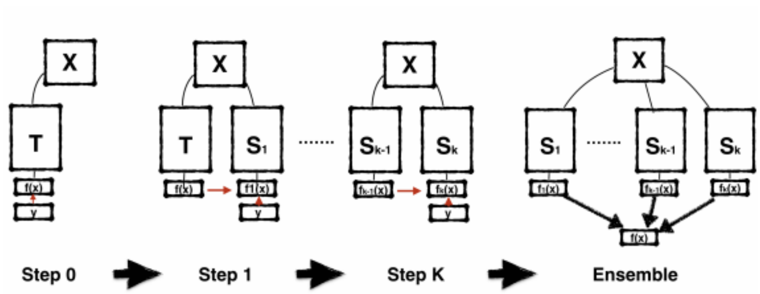

Figure 1: Illustration of the algorithm, taken from the paper [1]

The implementation as the previous method, has to be as generic as possible, so, to achieve the result, we will have in input:

- The master model
- The function to build the students in cascade
- The parameters to compile the models.

In this way there will be the maximum freedom on the choice of the student.

Listing 4: The code for training a list of students

```
def ban(teacher_model, n_students, build_model,
        train_dataset, valid_data, student_callbacks,
        epochs, batch_size,
        compile_args={'optimizer':'sgd',
                      'loss': 'categorical_crossentropy',
                      'metrics': ['accuracy']}):
  students = [build_model() for i in range(n_students)]
  students.insert(0, teacher_model)
  for student in students:
    student.compile(optimizer=compile_args['optimizer'], loss=compile_args['loss'],
                    metrics=compile_args['metrics'])
  history = []


  for i in range(1, len(students)):
    print("Training BAN-{}".format(i))
    current_history = distill_knowledge(students[i-1],
                                        students[i], train_dataset,
                                        valid_data, student_callbacks,
                                        epochs, batch_size, STEPS_PER_EPOCH)
    history.append(current_history)
  return history, students
```

2.3 Knowledge Distillation integrations

In one of the papers cited we can find an example of a different way to see knowledge distillation[2] Caruana and his collaborators have noticed that in some cases the lowest probabilities are so close to zero that they have almost no influence on the final result[6]. The strategy found is therefore to train the student on the logits of the master trying to minimizing the "mean squared error".
Another way to deal with the problem is to use a temperature variable in order to get a softened result, on which you can then effectively minimize crossentropy. So the soft-max with temperature T will be calculated in this way:

$$q_i = \frac{\exp(z_i/T)}{\sum_j(\exp(z_i/T)} \tag{1}$$

The paper[2] also introduces the possibility to use the ground truth to have a weighted average and get better results, also indicates that from empirical tests, assigning a lower weight to the ground truth produces better results.

2.4 Ensemble

Another technique mentioned in the paper[1] and which I think is worth replicating is the creation of an ensemble, obtained by the various generations of students. According to the researchers in several cases this technique allows for improvements in performance.
Let's see the code of the Ensemble:

Listing 5: The code for the BAN-Ensemble

```
class BANEnsemble(tf.keras.models.Model):
  def __init__(self, students, **kwargs):
    super(BANEnsemble, self).__init__(**kwargs)
    self.students = students
    self.len = len(students)

  def call(self, inputs):
    s_out = []
    for student in self.students:
      s_out.append(student(inputs))


    x = tf.keras.layers.Add()(s_out)
    x = layers.Lambda(lambda y: y/self.len)(x)
    return x
```

3 Preliminary Tests

Before starting the actual tests I will try to measure the performance of the model, all tests will be performed on cifar10 with data augmentation. In this paragraph I will just show how I performed the tests and the final results.
Let's start with the initialization of the datagen and the training of the master model.

Listing 6: The code for the Teacher training

```
BATCH_SIZE = 32
N_CLASSES = 10
STEPS_PER_EPOCH = len(x_train)//BATCH_SIZE


datagen = tf.keras.preprocessing.image.ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True
)
datagen.fit(x_train, seed=55)
train_data = datagen.flow(x_train, y_train, batch_size=BATCH_SIZE)

build_model= lambda: WideResidualNetwork(10, 28, 1)
teacher_callback = tf.keras.callbacks.EarlyStopping(patience=8, restore_best_weights=True)

teacher_model = build_model()
teacher_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
t_history = teacher_model.fit(train_data,
                    epochs=100,
                    steps_per_epoch=STEPS_PER_EPOCH,
                    callbacks = [teacher_callback],
                    validation_data=(x_valid, y_valid))
```

At this point it is necessary to start training students using ban. The test as in the paper will be performed for three students.

Listing 7: The code for the Students training

```
student_callback = tf.keras.callbacks.EarlyStopping(patience=12,
                restore_best_weights=True)
history, students = ban(teacher_model,3, build_model, train_data, (x_valid, y_valid),
 [student_callback], 100, BATCH_SIZE)
```

And finally I proceeded with the ensemble

Listing 8: The code for the Ensemble training

```
ban = BANEnsemble(students)
ban.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

The code to evaluate the model is the following:

Listing 9: The code for the Evaluation

```
print("Evaluating the students: ")
for student in students:
  student.evaluate(x_valid, y_valid)

print("Evaluating the ensemble: ")
ban.evaluate(x_valid, y_valid)
```

And this is the table with the results:

Table 1: Results on Cifar10

Metrics	**WResnet-28-1**	**BAN-1**	**BAN-2**	**BAN-3**	**Ensemble**
Accuracy	0.8570	0.8580	0.8574	0.8514	0.8747
Loss	0.4225	0.4213	0.4212	0.4424	0.3745

The performances do not seem noteworthy compared to the tests carried out on the same dataset, but it should be remembered that I did not have any shrewdness that researchers had instead, such as image augmentation or a tuning of the optimizer parameters. All this, however, was beyond the purposes of the report that instead aims to test the BANs on real datasets. So after some results that after all are encouraging, so I decided to continue with the next step.

4 Tests on Real Datasets

4.1 Binary Classification

The first real test, just to start with a warm-up, will focus on the detection of skin cancer. This is a binary classification, so it is likely that the ”dark knowledge” will have a lower weight. In any case I proceeded in the same way as the tests with our ”toy datasets”, so with a training dataset, a validation dataset and a test dataset. As in the previous version I used image augmentation on the training and an EarlyStopping Callback to avoid overfitting and keep the best results on the validation set.
The dataset can be found on kaggle at this url: https://www.kaggle.com/fanconic/skin-cancer-malignant-vs-benign

Listing 10: The code for the Dataset creation

```
train_datagen = tf.keras.preprocessing.image.ImageDataGenerator(
        rotation_range=10,
        zoom_range = 0.1,
        width_shift_range=0.2,
        height_shift_range=0.2)
valid_datagen = tf.keras.preprocessing.image.ImageDataGenerator(
    validation_split=0.1)

test_datagen = tf.keras.preprocessing.image.ImageDataGenerator()
train_generator = train_datagen.flow_from_directory(
        path_to_train,
        batch_size=BATCH_SIZE,
        class_mode='binary',
        seed=2,
        subset='training')
validation_generator = valid_datagen.flow_from_directory(
        path_to_train,
        batch_size=BATCH_SIZE,
        seed=2,
        class_mode='binary',
        subset='validation')
test_generator = test_datagen.flow_from_directory(
                path_to_test
        batch_size=BATCH_SIZE,
        class_mode='binary')
```

Note that I initialized two datasets on train with the same seed to avoid that the validation_set had image augmentation. But now let's see how to build, compile and train the master model.

Listing 11: The code for the Dataset creation

```
teacher_model = tf.keras.models.Sequential([
                        WideResidualNetwork(1, 28, 1, includeTop=False),
                        tf.keras.layers.Activation('sigmoid')
])
teacher_model.compile(optimizer='adam',
                                loss=tf.keras.losses.binary_crossentropy,
                                metrics=['accuracy'])
```

```
es_callback = tf.keras.callbacks.EarlyStopping(patience=15,
                                                                                          restore_best_w
history = teacher_model.fit(train_generator, callbacks=[es_callback],
                            steps_per_epoch=train_generator.samples//BATCH_SIZE,
                            validation_data=validation_generator,
                            validation_steps=validation_generator.samples//BATCH_SIZE,
                            epochs=150)
```

As in the preliminary tests I used Adam optimizer, because even though SGD is used in the study, Adam in practice seems to have very good results and less need for tuning.
So those are the result performed from the model in this particular task onto the test dataset:

Table 2: Results on Skin Cancer Dataset

Metrics	WResnet-28-1	BAN-1	BAN-2	BAN-3	Ensemble
Accuracy	0.8303	0.8091	0.8227	0.8514	0.8227
Loss	0.4225	0.4027	0.3864	0.3988	0.3681

4.2 Multiclass Classification

The dataset chosen for the multiclass classification is taken from kaggle at the following address: https://www.kaggle.com/gpiosenka/100-bird-species. It is a dataset with 225 different classes, quite uniform between the different classes and at the same time unbalanced within the classes between males and females, but it is suitable for the purpose.
In this section I will perform several tests, and all will have image augmentation. In order to avoid training time issues, all models will not have too many parameters and images will be resized with shape (96, 96, 3). But let's see the results now

4.2.1 BAN with equal teacher and student

In this section the reference model will be a WideResidualNetwork-28-2, we will train 3 generations of students, after which we will also see the results on the test dataset of an Ensemble containing all the previously trained networks. The training parameters are the same into the previous tests. Let's look the results

Table 3: Results on Skin Cancer Dataset

Metrics	WResnet-28-2	BAN-1	BAN-2	BAN-3	Ensemble
Accuracy	0.9280	0.9511	0.9336	0.8987	0.9662
Loss	0.2241	0.1955	0.4581	0.3558	0.1801

As you can see we have achieved significant improvements for the first two generations, and as you can see the Ensemble has the best performances even at the cost of a substantial increase in parameters.

References

[1] Tommaso Furlanello, Zachary C Lipton, Michael Tschannen, Laurent Itti, and Anima Anandkumar, Born again neural networks, arXiv preprint arXiv:1805.04770, (2018).

[2] Hinton, G., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network. arXiv:1503.02531, 2015.

[3] Yim, J., Joo, D., Bae, J., and Kim, J. A gift from knowledge distillation: Fast optimization, network minimization and transfer learning. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 7130 7138, 2017.

[4] Tan, S., Caruana, R., Hooker, G., and Gordo, A. Transparent model distillation. arXiv:1801.08640, 2018

[5] Zagoruyko, S. and Komodakis, N. Wide residual networks. In Proceedings of the British Machine Vision Conference (BMVC), pp. 87.1-87.12, 2016b.

[6] C. Bucilua, R. Caruana, and A. Niculescu-Mizil. Model compression. In Proceedings of the 12th ACM SIGKDD International Conference on Knowledge

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. CoRR, abs/1512.03385, 2015.