1. Deep Learning.
   1. Build a DNN with five hidden layers of 100 neurons each, He initialization, and the ELU activation function.
   2. Using Adam optimization and early stopping, try training it on MNIST but only on digits 0 to 4, as we will use transfer learning for digits 5 to 9 in the next exercise. You will need a softmax output layer with five neurons, and as always make sure to save checkpoints at regular intervals and save the final model so you can reuse it later.
   3. Tune the hyperparameters using cross-validation and see what precision you can achieve.
   4. Now try adding Batch Normalization and compare the learning curves: is it converging faster than before? Does it produce a better model?
   5. Is the model overfitting the training set? Try adding dropout to every layer and try again. Does it help?

Setup

First, let's make sure this notebook works well in both python 2 and 3, import a few common modules, ensure MatplotLib plots figures inline and prepare a function to save the figures:

In [1]:

# To support both python 2 and python 3

from \_\_future\_\_ import division, print\_function, unicode\_literals

# Common imports

import numpy as np

import os

# to make this notebook's output stable across runs

def reset\_graph(seed=42):

tf.reset\_default\_graph()

tf.set\_random\_seed(seed)

np.random.seed(seed)

# To plot pretty figures

%matplotlib inline

import matplotlib

import matplotlib.pyplot as plt

plt.rcParams['axes.labelsize'] = 14

plt.rcParams['xtick.labelsize'] = 12

plt.rcParams['ytick.labelsize'] = 12

# Where to save the figures

PROJECT\_ROOT\_DIR = "."

CHAPTER\_ID = "deep"

def save\_fig(fig\_id, tight\_layout=True):

path = os.path.join(PROJECT\_ROOT\_DIR, "images", CHAPTER\_ID, fig\_id + ".png")

print("Saving figure", fig\_id)

if tight\_layout:

plt.tight\_layout()

plt.savefig(path, format='png', dpi=300)

Vanishing/Exploding Gradients Problem

In [2]:

def logit(z):

return 1 / (1 + np.exp(-z))

In [3]:

z = np.linspace(-5, 5, 200)

plt.plot([-5, 5], [0, 0], 'k-')

plt.plot([-5, 5], [1, 1], 'k--')

plt.plot([0, 0], [-0.2, 1.2], 'k-')

plt.plot([-5, 5], [-3/4, 7/4], 'g--')

plt.plot(z, logit(z), "b-", linewidth=2)

props = dict(facecolor='black', shrink=0.1)

plt.annotate('Saturating', xytext=(3.5, 0.7), xy=(5, 1), arrowprops=props, fontsize=14, ha="center")

plt.annotate('Saturating', xytext=(-3.5, 0.3), xy=(-5, 0), arrowprops=props, fontsize=14, ha="center")

plt.annotate('Linear', xytext=(2, 0.2), xy=(0, 0.5), arrowprops=props, fontsize=14, ha="center")

plt.grid(True)

plt.title("Sigmoid activation function", fontsize=14)

plt.axis([-5, 5, -0.2, 1.2])

save\_fig("sigmoid\_saturation\_plot")

plt.show()

Saving figure sigmoid\_saturation\_plot

Xavier and He Initialization

Note: the book uses tensorflow.contrib.layers.fully\_connected() rather than tf.layers.dense() (which did not exist when this chapter was written). It is now preferable to use tf.layers.dense(), because anything in the contrib module may change or be deleted without notice. The dense() function is almost identical to the fully\_connected() function. The main differences relevant to this chapter are:

several parameters are renamed: scope becomes name, activation\_fn becomes activation (and similarly the \_fn suffix is removed from other parameters such as normalizer\_fn), weights\_initializer becomes kernel\_initializer, etc.

the default activation is now None rather than tf.nn.relu.

it does not support tensorflow.contrib.framework.arg\_scope() (introduced later in chapter 11).

it does not support regularizer params (introduced later in chapter 11).

In [4]:

import tensorflow as tf

In [5]:

reset\_graph()

n\_inputs = 28 \* 28 # MNIST

n\_hidden1 = 300

X = tf.placeholder(tf.float32, shape=(None, n\_inputs), name="X")

In [6]:

he\_init = tf.variance\_scaling\_initializer()

hidden1 = tf.layers.dense(X, n\_hidden1, activation=tf.nn.relu,

kernel\_initializer=he\_init, name="hidden1")

Nonsaturating Activation Functions

Leaky ReLU

In [7]:

def leaky\_relu(z, alpha=0.01):

return np.maximum(alpha\*z, z)

In [8]:

plt.plot(z, leaky\_relu(z, 0.05), "b-", linewidth=2)

plt.plot([-5, 5], [0, 0], 'k-')

plt.plot([0, 0], [-0.5, 4.2], 'k-')

plt.grid(True)

props = dict(facecolor='black', shrink=0.1)

plt.annotate('Leak', xytext=(-3.5, 0.5), xy=(-5, -0.2), arrowprops=props, fontsize=14, ha="center")

plt.title("Leaky ReLU activation function", fontsize=14)

plt.axis([-5, 5, -0.5, 4.2])

save\_fig("leaky\_relu\_plot")

plt.show()

Saving figure leaky\_relu\_plot

Implementing Leaky ReLU in TensorFlow:

In [9]:

reset\_graph()

X = tf.placeholder(tf.float32, shape=(None, n\_inputs), name="X")

In [10]:

def leaky\_relu(z, name=None):

return tf.maximum(0.01 \* z, z, name=name)

hidden1 = tf.layers.dense(X, n\_hidden1, activation=leaky\_relu, name="hidden1")

Let's train a neural network on MNIST using the Leaky ReLU. First let's create the graph:

In [11]:

reset\_graph()

n\_inputs = 28 \* 28 # MNIST

n\_hidden1 = 300

n\_hidden2 = 100

n\_outputs = 10

In [12]:

X = tf.placeholder(tf.float32, shape=(None, n\_inputs), name="X")

y = tf.placeholder(tf.int32, shape=(None), name="y")

In [13]:

with tf.name\_scope("dnn"):

hidden1 = tf.layers.dense(X, n\_hidden1, activation=leaky\_relu, name="hidden1")

hidden2 = tf.layers.dense(hidden1, n\_hidden2, activation=leaky\_relu, name="hidden2")

logits = tf.layers.dense(hidden2, n\_outputs, name="outputs")

In [14]:

with tf.name\_scope("loss"):

xentropy = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=y, logits=logits)

loss = tf.reduce\_mean(xentropy, name="loss")

In [15]:

learning\_rate = 0.01

with tf.name\_scope("train"):

optimizer = tf.train.GradientDescentOptimizer(learning\_rate)

training\_op = optimizer.minimize(loss)

In [16]:

with tf.name\_scope("eval"):

correct = tf.nn.in\_top\_k(logits, y, 1)

accuracy = tf.reduce\_mean(tf.cast(correct, tf.float32))

In [17]:

init = tf.global\_variables\_initializer()

saver = tf.train.Saver()

Let's load the data:

Warning: tf.examples.tutorials.mnist is deprecated. We will use tf.keras.datasets.mnist instead.

In [18]:

(X\_train, y\_train), (X\_test, y\_test) = tf.keras.datasets.mnist.load\_data()

X\_train = X\_train.astype(np.float32).reshape(-1, 28\*28) / 255.0

X\_test = X\_test.astype(np.float32).reshape(-1, 28\*28) / 255.0

y\_train = y\_train.astype(np.int32)

y\_test = y\_test.astype(np.int32)

X\_valid, X\_train = X\_train[:5000], X\_train[5000:]

y\_valid, y\_train = y\_train[:5000], y\_train[5000:]

In [19]:

def shuffle\_batch(X, y, batch\_size):

rnd\_idx = np.random.permutation(len(X))

n\_batches = len(X) // batch\_size

for batch\_idx in np.array\_split(rnd\_idx, n\_batches):

X\_batch, y\_batch = X[batch\_idx], y[batch\_idx]

yield X\_batch, y\_batch

In [20]:

n\_epochs = 40

batch\_size = 50

with tf.Session() as sess:

init.run()

for epoch in range(n\_epochs):

for X\_batch, y\_batch in shuffle\_batch(X\_train, y\_train, batch\_size):

sess.run(training\_op, feed\_dict={X: X\_batch, y: y\_batch})

if epoch % 5 == 0:

acc\_batch = accuracy.eval(feed\_dict={X: X\_batch, y: y\_batch})

acc\_valid = accuracy.eval(feed\_dict={X: X\_valid, y: y\_valid})

print(epoch, "Batch accuracy:", acc\_batch, "Validation accuracy:", acc\_valid)

save\_path = saver.save(sess, "./my\_model\_final.ckpt")

0 Batch accuracy: 0.86 Validation accuracy: 0.9044

5 Batch accuracy: 0.94 Validation accuracy: 0.9496

10 Batch accuracy: 0.92 Validation accuracy: 0.9654

15 Batch accuracy: 0.94 Validation accuracy: 0.971

20 Batch accuracy: 1.0 Validation accuracy: 0.9764

25 Batch accuracy: 1.0 Validation accuracy: 0.9778

30 Batch accuracy: 0.98 Validation accuracy: 0.978

35 Batch accuracy: 1.0 Validation accuracy: 0.9788

ELU

In [21]:

def elu(z, alpha=1):

return np.where(z < 0, alpha \* (np.exp(z) - 1), z)

In [22]:

plt.plot(z, elu(z), "b-", linewidth=2)

plt.plot([-5, 5], [0, 0], 'k-')

plt.plot([-5, 5], [-1, -1], 'k--')

plt.plot([0, 0], [-2.2, 3.2], 'k-')

plt.grid(True)

plt.title(r"ELU activation function ($\alpha=1$)", fontsize=14)

plt.axis([-5, 5, -2.2, 3.2])

save\_fig("elu\_plot")

plt.show()

Saving figure elu\_plot

Implementing ELU in TensorFlow is trivial, just specify the activation function when building each layer:

In [23]:

reset\_graph()

X = tf.placeholder(tf.float32, shape=(None, n\_inputs), name="X")

In [24]:

hidden1 = tf.layers.dense(X, n\_hidden1, activation=tf.nn.elu, name="hidden1")

SELU

This activation function was proposed in this [great paper](https://arxiv.org/pdf/1706.02515.pdf) by Günter Klambauer, Thomas Unterthiner and Andreas Mayr, published in June 2017. During training, a neural network composed exclusively of a stack of dense layers using the SELU activation function and LeCun initialization will self-normalize: the output of each layer will tend to preserve the same mean and variance during training, which solves the vanishing/exploding gradients problem. As a result, this activation function outperforms the other activation functions very significantly for such neural nets, so you should really try it out. Unfortunately, the self-normalizing property of the SELU activation function is easily broken: you cannot use ℓ1 or ℓ2 regularization, regular dropout, max-norm, skip connections or other non-sequential topologies (so recurrent neural networks won't self-normalize). However, in practice it works quite well with sequential CNNs. If you break self-normalization, SELU will not necessarily outperform other activation functions.

In [25]:

from scipy.special import erfc

# alpha and scale to self normalize with mean 0 and standard deviation 1

# (see equation 14 in the paper):

alpha\_0\_1 = -np.sqrt(2 / np.pi) / (erfc(1/np.sqrt(2)) \* np.exp(1/2) - 1)

scale\_0\_1 = (1 - erfc(1 / np.sqrt(2)) \* np.sqrt(np.e)) \* np.sqrt(2 \* np.pi) \* (2 \* erfc(np.sqrt(2))\*np.e\*\*2 + np.pi\*erfc(1/np.sqrt(2))\*\*2\*np.e - 2\*(2+np.pi)\*erfc(1/np.sqrt(2))\*np.sqrt(np.e)+np.pi+2)\*\*(-1/2)

In [26]:

def selu(z, scale=scale\_0\_1, alpha=alpha\_0\_1):

return scale \* elu(z, alpha)

In [27]:

plt.plot(z, selu(z), "b-", linewidth=2)

plt.plot([-5, 5], [0, 0], 'k-')

plt.plot([-5, 5], [-1.758, -1.758], 'k--')

plt.plot([0, 0], [-2.2, 3.2], 'k-')

plt.grid(True)

plt.title(r"SELU activation function", fontsize=14)

plt.axis([-5, 5, -2.2, 3.2])

save\_fig("selu\_plot")

plt.show()

Saving figure selu\_plot

By default, the SELU hyperparameters (scale and alpha) are tuned in such a way that the mean output of each neuron remains close to 0, and the standard deviation remains close to 1 (assuming the inputs are standardized with mean 0 and standard deviation 1 too). Using this activation function, even a 1,000 layer deep neural network preserves roughly mean 0 and standard deviation 1 across all layers, avoiding the exploding/vanishing gradients problem:

In [28]:

np.random.seed(42)

Z = np.random.normal(size=(500, 100)) # standardized inputs

for layer in range(1000):

W = np.random.normal(size=(100, 100), scale=np.sqrt(1 / 100)) # LeCun initialization

Z = selu(np.dot(Z, W))

means = np.mean(Z, axis=0).mean()

stds = np.std(Z, axis=0).mean()

if layer % 100 == 0:

print("Layer {}: mean {:.2f}, std deviation {:.2f}".format(layer, means, stds))

Layer 0: mean -0.00, std deviation 1.00

Layer 100: mean 0.02, std deviation 0.96

Layer 200: mean 0.01, std deviation 0.90

Layer 300: mean -0.02, std deviation 0.92

Layer 400: mean 0.05, std deviation 0.89

Layer 500: mean 0.01, std deviation 0.93

Layer 600: mean 0.02, std deviation 0.92

Layer 700: mean -0.02, std deviation 0.90

Layer 800: mean 0.05, std deviation 0.83

Layer 900: mean 0.02, std deviation 1.00

The tf.nn.selu() function was added in TensorFlow 1.4. For earlier versions, you can use the following implementation:

In [29]:

def selu(z, scale=alpha\_0\_1, alpha=scale\_0\_1):

return scale \* tf.where(z >= 0.0, z, alpha \* tf.nn.elu(z))

However, the SELU activation function cannot be used along with regular Dropout (this would cancel the SELU activation function's self-normalizing property). Fortunately, there is a Dropout variant called Alpha Dropout proposed in the same paper. It is available in tf.contrib.nn.alpha\_dropout() since TF 1.4 (or check out [this implementation](https://github.com/bioinf-jku/SNNs/blob/master/selu.py) by the Institute of Bioinformatics, Johannes Kepler University Linz).

Let's create a neural net for MNIST using the SELU activation function:

In [30]:

reset\_graph()

n\_inputs = 28 \* 28 # MNIST

n\_hidden1 = 300

n\_hidden2 = 100

n\_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n\_inputs), name="X")

y = tf.placeholder(tf.int32, shape=(None), name="y")

with tf.name\_scope("dnn"):

hidden1 = tf.layers.dense(X, n\_hidden1, activation=selu, name="hidden1")

hidden2 = tf.layers.dense(hidden1, n\_hidden2, activation=selu, name="hidden2")

logits = tf.layers.dense(hidden2, n\_outputs, name="outputs")

with tf.name\_scope("loss"):

xentropy = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=y, logits=logits)

loss = tf.reduce\_mean(xentropy, name="loss")

learning\_rate = 0.01

with tf.name\_scope("train"):

optimizer = tf.train.GradientDescentOptimizer(learning\_rate)

training\_op = optimizer.minimize(loss)

with tf.name\_scope("eval"):

correct = tf.nn.in\_top\_k(logits, y, 1)

accuracy = tf.reduce\_mean(tf.cast(correct, tf.float32))

init = tf.global\_variables\_initializer()

saver = tf.train.Saver()

n\_epochs = 40

batch\_size = 50

Now let's train it. Do not forget to scale the inputs to mean 0 and standard deviation 1:

In [31]:

means = X\_train.mean(axis=0, keepdims=True)

stds = X\_train.std(axis=0, keepdims=True) + 1e-10

X\_val\_scaled = (X\_valid - means) / stds

with tf.Session() as sess:

init.run()

for epoch in range(n\_epochs):

for X\_batch, y\_batch in shuffle\_batch(X\_train, y\_train, batch\_size):

X\_batch\_scaled = (X\_batch - means) / stds

sess.run(training\_op, feed\_dict={X: X\_batch\_scaled, y: y\_batch})

if epoch % 5 == 0:

acc\_batch = accuracy.eval(feed\_dict={X: X\_batch\_scaled, y: y\_batch})

acc\_valid = accuracy.eval(feed\_dict={X: X\_val\_scaled, y: y\_valid})

print(epoch, "Batch accuracy:", acc\_batch, "Validation accuracy:", acc\_valid)

save\_path = saver.save(sess, "./my\_model\_final\_selu.ckpt")

0 Batch accuracy: 0.88 Validation accuracy: 0.923

5 Batch accuracy: 0.98 Validation accuracy: 0.9578

10 Batch accuracy: 1.0 Validation accuracy: 0.9664

15 Batch accuracy: 0.96 Validation accuracy: 0.9682

20 Batch accuracy: 1.0 Validation accuracy: 0.9694

25 Batch accuracy: 1.0 Validation accuracy: 0.9688

30 Batch accuracy: 1.0 Validation accuracy: 0.9694

35 Batch accuracy: 1.0 Validation accuracy: 0.97

Batch Normalization

Note: the book uses tensorflow.contrib.layers.batch\_norm() rather than tf.layers.batch\_normalization() (which did not exist when this chapter was written). It is now preferable to use tf.layers.batch\_normalization(), because anything in the contrib module may change or be deleted without notice. Instead of using the batch\_norm() function as a regularizer parameter to the fully\_connected() function, we now use batch\_normalization() and we explicitly create a distinct layer. The parameters are a bit different, in particular:

decay is renamed to momentum,

is\_training is renamed to training,

updates\_collections is removed: the update operations needed by batch normalization are added to the UPDATE\_OPS collection and you need to explicity run these operations during training (see the execution phase below),

we don't need to specify scale=True, as that is the default.

Also note that in order to run batch norm just before each hidden layer's activation function, we apply the ELU activation function manually, right after the batch norm layer.

Note: since the tf.layers.dense() function is incompatible with tf.contrib.layers.arg\_scope() (which is used in the book), we now use python's functools.partial() function instead. It makes it easy to create a my\_dense\_layer() function that just calls tf.layers.dense() with the desired parameters automatically set (unless they are overridden when calling my\_dense\_layer()). As you can see, the code remains very similar.

In [32]:

reset\_graph()

import tensorflow as tf

n\_inputs = 28 \* 28

n\_hidden1 = 300

n\_hidden2 = 100

n\_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n\_inputs), name="X")

training = tf.placeholder\_with\_default(False, shape=(), name='training')

hidden1 = tf.layers.dense(X, n\_hidden1, name="hidden1")

bn1 = tf.layers.batch\_normalization(hidden1, training=training, momentum=0.9)

bn1\_act = tf.nn.elu(bn1)

hidden2 = tf.layers.dense(bn1\_act, n\_hidden2, name="hidden2")

bn2 = tf.layers.batch\_normalization(hidden2, training=training, momentum=0.9)

bn2\_act = tf.nn.elu(bn2)

logits\_before\_bn = tf.layers.dense(bn2\_act, n\_outputs, name="outputs")

logits = tf.layers.batch\_normalization(logits\_before\_bn, training=training,

momentum=0.9)

In [33]:

reset\_graph()

X = tf.placeholder(tf.float32, shape=(None, n\_inputs), name="X")

training = tf.placeholder\_with\_default(False, shape=(), name='training')

To avoid repeating the same parameters over and over again, we can use Python's partial() function:

In [34]:

from functools import partial

my\_batch\_norm\_layer = partial(tf.layers.batch\_normalization,

training=training, momentum=0.9)

hidden1 = tf.layers.dense(X, n\_hidden1, name="hidden1")

bn1 = my\_batch\_norm\_layer(hidden1)

bn1\_act = tf.nn.elu(bn1)

hidden2 = tf.layers.dense(bn1\_act, n\_hidden2, name="hidden2")

bn2 = my\_batch\_norm\_layer(hidden2)

bn2\_act = tf.nn.elu(bn2)

logits\_before\_bn = tf.layers.dense(bn2\_act, n\_outputs, name="outputs")

logits = my\_batch\_norm\_layer(logits\_before\_bn)

Let's build a neural net for MNIST, using the ELU activation function and Batch Normalization at each layer:

In [35]:

reset\_graph()

batch\_norm\_momentum = 0.9

X = tf.placeholder(tf.float32, shape=(None, n\_inputs), name="X")

y = tf.placeholder(tf.int32, shape=(None), name="y")

training = tf.placeholder\_with\_default(False, shape=(), name='training')

with tf.name\_scope("dnn"):

he\_init = tf.variance\_scaling\_initializer()

my\_batch\_norm\_layer = partial(

tf.layers.batch\_normalization,

training=training,

momentum=batch\_norm\_momentum)

my\_dense\_layer = partial(

tf.layers.dense,

kernel\_initializer=he\_init)

hidden1 = my\_dense\_layer(X, n\_hidden1, name="hidden1")

bn1 = tf.nn.elu(my\_batch\_norm\_layer(hidden1))

hidden2 = my\_dense\_layer(bn1, n\_hidden2, name="hidden2")

bn2 = tf.nn.elu(my\_batch\_norm\_layer(hidden2))

logits\_before\_bn = my\_dense\_layer(bn2, n\_outputs, name="outputs")

logits = my\_batch\_norm\_layer(logits\_before\_bn)

with tf.name\_scope("loss"):

xentropy = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=y, logits=logits)

loss = tf.reduce\_mean(xentropy, name="loss")

with tf.name\_scope("train"):

optimizer = tf.train.GradientDescentOptimizer(learning\_rate)

training\_op = optimizer.minimize(loss)

with tf.name\_scope("eval"):

correct = tf.nn.in\_top\_k(logits, y, 1)

accuracy = tf.reduce\_mean(tf.cast(correct, tf.float32))

init = tf.global\_variables\_initializer()

saver = tf.train.Saver()

Note: since we are using tf.layers.batch\_normalization() rather than tf.contrib.layers.batch\_norm() (as in the book), we need to explicitly run the extra update operations needed by batch normalization (sess.run([training\_op, extra\_update\_ops],...).

In [36]:

n\_epochs = 20

batch\_size = 200

In [37]:

extra\_update\_ops = tf.get\_collection(tf.GraphKeys.UPDATE\_OPS)

with tf.Session() as sess:

init.run()

for epoch in range(n\_epochs):

for X\_batch, y\_batch in shuffle\_batch(X\_train, y\_train, batch\_size):

sess.run([training\_op, extra\_update\_ops],

feed\_dict={training: True, X: X\_batch, y: y\_batch})

accuracy\_val = accuracy.eval(feed\_dict={X: X\_valid, y: y\_valid})

print(epoch, "Validation accuracy:", accuracy\_val)

save\_path = saver.save(sess, "./my\_model\_final.ckpt")

0 Validation accuracy: 0.8952

1 Validation accuracy: 0.9202

2 Validation accuracy: 0.9318

3 Validation accuracy: 0.9422

4 Validation accuracy: 0.9468

5 Validation accuracy: 0.954

6 Validation accuracy: 0.9568

7 Validation accuracy: 0.96

8 Validation accuracy: 0.962

9 Validation accuracy: 0.9638

10 Validation accuracy: 0.9662

11 Validation accuracy: 0.9682

12 Validation accuracy: 0.9672

13 Validation accuracy: 0.9696

14 Validation accuracy: 0.9706

15 Validation accuracy: 0.9704

16 Validation accuracy: 0.9718

17 Validation accuracy: 0.9726

18 Validation accuracy: 0.9738

19 Validation accuracy: 0.9742

What!? That's not a great accuracy for MNIST. Of course, if you train for longer it will get much better accuracy, but with such a shallow network, Batch Norm and ELU are unlikely to have very positive impact: they shine mostly for much deeper nets.

Note that you could also make the training operation depend on the update operations:

with tf.name\_scope("train"):

optimizer = tf.train.GradientDescentOptimizer(learning\_rate)

extra\_update\_ops = tf.get\_collection(tf.GraphKeys.UPDATE\_OPS)

with tf.control\_dependencies(extra\_update\_ops):

training\_op = optimizer.minimize(loss)

This way, you would just have to evaluate the training\_op during training, TensorFlow would automatically run the update operations as well:

sess.run(training\_op, feed\_dict={training: True, X: X\_batch, y: y\_batch})

One more thing: notice that the list of trainable variables is shorter than the list of all global variables. This is because the moving averages are non-trainable variables. If you want to reuse a pretrained neural network (see below), you must not forget these non-trainable variables.

In [38]:

[v.name for v in tf.trainable\_variables()]

Out[38]:

['hidden1/kernel:0',

'hidden1/bias:0',

'batch\_normalization/gamma:0',

'batch\_normalization/beta:0',

'hidden2/kernel:0',

'hidden2/bias:0',

'batch\_normalization\_1/gamma:0',

'batch\_normalization\_1/beta:0',

'outputs/kernel:0',

'outputs/bias:0',

'batch\_normalization\_2/gamma:0',

'batch\_normalization\_2/beta:0']

In [39]:

[v.name for v in tf.global\_variables()]

Out[39]:

['hidden1/kernel:0',

'hidden1/bias:0',

'batch\_normalization/gamma:0',

'batch\_normalization/beta:0',

'batch\_normalization/moving\_mean:0',

'batch\_normalization/moving\_variance:0',

'hidden2/kernel:0',

'hidden2/bias:0',

'batch\_normalization\_1/gamma:0',

'batch\_normalization\_1/beta:0',

'batch\_normalization\_1/moving\_mean:0',

'batch\_normalization\_1/moving\_variance:0',

'outputs/kernel:0',

'outputs/bias:0',

'batch\_normalization\_2/gamma:0',

'batch\_normalization\_2/beta:0',

'batch\_normalization\_2/moving\_mean:0',

'batch\_normalization\_2/moving\_variance:0']

2.Transfer learning.

* 1. Create a new DNN that reuses all the pretrained hidden layers of the previous model, freezes them, and replaces the softmax output layer with a new one.
  2. Train this new DNN on digits 5 to 9, using only 100 images per digit, and time how long it takes. Despite this small number of examples, can you achieve high precision?
  3. Try caching the frozen layers, and train the model again: how much faster is it now?
  4. Try again reusing just four hidden layers instead of five. Can you achieve a higher precision?
  5. Now unfreeze the top two hidden layers and continue training: can you get the model to perform even better?

Transfer learning from pre-trained models

How to solve any image classification problem quickly and easily

Deep learning is fast becoming a key instrument in artificial intelligence applications (LeCun et al. 2015). For example, in areas such as computer vision, natural language processing, and speech recognition, deep learning has been producing remarkable results. Therefore, there is a growing interest in deep learning.

One of the problems where deep learning excels is image classification (Rawat & Wang 2017). The goal in image classification is to classify a specific picture according to a set of possible categories. A classic example of image classification is the identification of cats and dogs in a set of pictures (e.g. [Dogs vs. Cats Kaggle Competition](https://www.kaggle.com/c/dogs-vs-cats)).

From a deep learning perspective, the image classification problem can be solved through transfer learning. Actually, several state-of-the-art results in image classification are based on transfer learning solutions (Krizhevsky et al. 2012, Simonyan & Zisserman 2014, He et al. 2016). A comprehensive review on transfer learning is provided by Pan & Yang (2010).

This article shows how to implement a transfer learning solution for image classification problems. The implementation proposed in this article is based on Keras (Chollet 2015), which uses the programming language Python. Following this implementation, you will be able to solve any image classification problem quickly and easily.

The article has been organised in the following way:

Transfer learning

Convolutional neural networks

Repurposing a pre-trained model

Transfer learning process

Classifiers on top of deep convolutional neural networks

Example

Summary

References

1. Transfer learning

Transfer learning is a popular method in computer vision because it allows us to build accurate models in a timesaving way (Rawat & Wang 2017). With transfer learning, instead of starting the learning process from scratch, you start from patterns that have been learned when solving a different problem. This way you leverage previous learnings and avoid starting from scratch. Take it as the deep learning version of [Chartres](https://en.wikipedia.org/wiki/Standing_on_the_shoulders_of_giants)’ expression ‘standing on the shoulder of giants’.

In computer vision, transfer learning is usually expressed through the use of pre-trained models. A pre-trained model is a model that was trained on a large benchmark dataset to solve a problem similar to the one that we want to solve. Accordingly, due to the computational cost of training such models, it is common practice to import and use models from published literature (e.g. [VGG](https://arxiv.org/pdf/1409.1556.pdf), [Inception](https://arxiv.org/pdf/1512.00567.pdf), [MobileNet](https://arxiv.org/pdf/1704.04861.pdf" \t "_blank)). A comprehensive review of pre-trained models’ performance on computer vision problems using data from the ImageNet (Deng et al. 2009) challenge is presented by Canziani et al. (2016).

2. Convolutional neural networks

Several pre-trained models used in transfer learning are based on large convolutional neural networks (CNN) (Voulodimos et al. 2018). In general, CNN was shown to excel in a wide range of computer vision tasks (Bengio 2009). Its high performance and its easiness in training are two of the main factors driving the popularity of CNN over the last years.

A typical CNN has two parts:

Convolutional base, which is composed by a stack of convolutional and pooling layers. The main goal of the convolutional base is to generate features from the image. For an intuitive explanation of convolutional and pooling layers, please refer to Chollet (2017).

Classifier, which is usually composed by fully connected layers. The main goal of the classifier is to classify the image based on the detected features. A fully connected layer is a layer whose neurons have full connections to all activation in the previous layer.

One important aspect of these deep learning models is that they can automatically learn hierarchical feature representations. This means that features computed by the first layer are general and can be reused in different problem domains, while features computed by the last layer are specific and depend on the chosen dataset and task. According to Yosinski et al. (2014), ‘if first-layer features are general and last-layer features are specific, then there must be a transition from general to specific somewhere in the network’. As a result, the convolutional base of our CNN — especially its lower layers (those who are closer to the inputs) — refer to general features, whereas the classifier part, and some of the higher layers of the convolutional base, refer to specialised features.

3. Repurposing a pre-trained model

When you’re repurposing a pre-trained model for your own needs, you start by removing the original classifier, then you add a new classifier that fits your purposes, and finally you have to fine-tune your model according to one of three strategies:

Train the entire model. In this case, you use the architecture of the pre-trained model and train it according to your dataset. You’re learning the model from scratch, so you’ll need a large dataset (and a lot of computational power).

Train some layers and leave the others frozen. As you remember, lower layers refer to general features (problem independent), while higher layers refer to specific features (problem dependent). Here, we play with that dichotomy by choosing how much we want to adjust the weights of the network (a frozen layer does not change during training). Usually, if you’ve a small dataset and a large number of parameters, you’ll leave more layers frozen to avoid overfitting. By contrast, if the dataset is large and the number of parameters is small, you can improve your model by training more layers to the new task since overfitting is not an issue.

Freeze the convolutional base. This case corresponds to an extreme situation of the train/freeze trade-off. The main idea is to keep the convolutional base in its original form and then use its outputs to feed the classifier. You’re using the pre-trained model as a fixed feature extraction mechanism, which can be useful if you’re short on computational power, your dataset is small, and/or pre-trained model solves a problem very similar to the one you want to solve.

Unlike Strategy 3, whose application is straightforward, Strategy 1 and Strategy 2 require you to be careful with the learning rate used in the convolutional part. The learning rate is a hyper-parameter that controls how much you adjust the weights of your network. When you’re using a pre-trained model based on CNN, it’s smart to use a small learning rate because high learning rates increase the risk of losing previous knowledge. Assuming that the pre-trained model has been well trained, which is a fair assumption, keeping a small learning rate will ensure that you don’t distort the CNN weights too soon and too much.

4. Transfer learning process

From a practical perspective, the entire transfer learning process can be summarised as follows:

Select a pre-trained model. From the wide range of pre-trained models that are available, you pick one that looks suitable for your problem. For example, if you’re using Keras, you immediately have access to a set of models, such as VGG (Simonyan & Zisserman 2014), InceptionV3 (Szegedy et al. 2015), and ResNet5 (He et al. 2015). [Here](https://keras.io/applications/) you can see all the models available on Keras.

Classify your problem according to the Size-Similarity Matrix. In Figure 3 you have ‘The Matrix’ that controls your choices. This matrix classifies your computer vision problem considering the size of your dataset and its similarity to the dataset in which your pre-trained model was trained. As a rule of thumb, consider that your dataset is small if it has less than 1000 images per class. Regarding dataset similarity, let common sense prevail. For example, if your task is to identify cats and dogs, ImageNet would be a similar dataset because it has images of cats and dogs. However, if your task is to identify cancer cells, ImageNet can’t be considered a similar dataset.

Fine-tune your model. Here you can use the Size-Similarity Matrix to guide your choice and then refer to the three options we mentioned before about repurposing a pre-trained model. Figure 4 provides a visual summary of the text that follows.

Quadrant 1. Large dataset, but different from the pre-trained model’s dataset. This situation will lead you to Strategy 1. Since you have a large dataset, you’re able to train a model from scratch and do whatever you want. Despite the dataset dissimilarity, in practice, it can still be useful to initialise your model from a pre-trained model, using its architecture and weights.

Quadrant 2. Large dataset and similar to the pre-trained model’s dataset. Here you’re in la-la land. Any option works. Probably, the most efficient option is Strategy 2. Since we have a large dataset, overfitting shouldn’t be an issue, so we can learn as much as we want. However, since the datasets are similar, we can save ourselves from a huge training effort by leveraging previous knowledge. Therefore, it should be enough to train the classifier and the top layers of the convolutional base.

Quadrant 3. Small dataset and different from the pre-trained model’s dataset. This is the 2–7 off-suit hand of computer vision problems. Everything is against you. If complaining is not an option, the only hope you have is Strategy 2. It will be hard to find a balance between the number of layers to train and freeze. If you go to deep your model can overfit, if you stay in the shallow end of your model you won’t learn anything useful. Probably, you’ll need to go deeper than in Quadrant 2 and you’ll need to consider data augmentation techniques (a nice summary on data augmentation techniques is provided [here](https://medium.com/nanonets/how-to-use-deep-learning-when-you-have-limited-data-part-2-data-augmentation-c26971dc8ced)).

Quadrant 4. Small dataset, but similar to the pre-trained model’s dataset. I asked Master Yoda about this one he told me that ‘be the best option, Strategy 3 should’. I don’t know about you, but I don’t underestimate the Force. Accordingly, go for Strategy 3. You just need to remove the last fully-connected layer (output layer), run the pre-trained model as a fixed feature extractor, and then use the resulting features to train a new classifier.

3.Pretraining on an auxiliary task.

* 1. In this exercise you will build a DNN that compares two MNIST digit images and predicts whether they represent the same digit or not. Then you will reuse the lower layers of this network to train an MNIST classifier using very little training data. Start by building two DNNs (let’s call them DNN A and B), both similar to the one you built earlier but without the output layer: each DNN should have five hidden layers of 100 neurons each, He initialization, and ELU activation. Next, add one more hidden layer with 10 units on top of both DNNs. To do this, you should use TensorFlow’s concat() function with axis=1 to concatenate the outputs of both DNNs for each instance, then feed the result to the hidden layer. Finally, add an output layer with a single neuron using the logistic activation function.
  2. Split the MNIST training set in two sets: split #1 should containing 55,000 images, and split #2 should contain contain 5,000 images. Create a function that generates a training batch where each instance is a pair of MNIST images picked from split #1. Half of the training instances should be pairs of images that belong to the same class, while the other half should be images from different classes. For each pair, the training label should be 0 if the images are from the same class, or 1 if they are from different classes.
  3. Train the DNN on this training set. For each image pair, you can simultaneously feed the first image to DNN A and the second image to DNN B. The whole network will gradually learn to tell whether two images belong to the same class or not.
  4. Now create a new DNN by reusing and freezing the hidden layers of DNN A and adding a softmax output layer on top with 10 neurons. Train this network on split #2 and see if you can achieve high performance despite having only 500 images per class.

Learning through Auxiliary Task

I’d like to illustrate how auxiliary tasks can be beneficial for machine learning. I’ll first define this notion, provide examples and show how these tasks are usually incorporated in the mathematical formulation of deep learning problems. I’ll then propose an alternative approach and present first experimental results for this approach.

[Auxiliary tasks in machine learning](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#auxiliary-tasks-in-machine-learning)

[Standard approach for deep auxiliary learning](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#standard-approach-for-deep-auxiliary-learning)

[An alternative approach](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#an-alternative-approach)

[Comparison to unweighted and weighted cosine](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#comparison-to-unweighted-and-weighted-cosine)

[Experiments](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#experiments)

[Toy experiments](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#toy-experiments)

[Experiments on a real dataset](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#experiments-on-a-real-dataset)

[Experiments on a synthetic dataset](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#experiments-on-a-synthetic-dataset)

[Conclusion](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#conclusion)

[References](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#references)

Auxiliary tasks in machine learning

In machine learning, auxiliary tasks are tasks we try to accomplish with the sole objective of better performing one or several primary tasks. This situation, referred to here as auxiliary learning, contrasts with multitask learning[1](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#fn:1), for which we are genuinely interested in accomplishing well all tasks, and single task learning, for which only one task is considered.

|  |  |  |
| --- | --- | --- |
|  | Tasks performed during training | Tasks considered when assessing performance |
| Single task learning | One task | |
| Multitask learning | Several tasks | |
| Auxiliary learning | One or several primary tasks, one or several auxiliary tasks | Primary tasks |

Table 1. Auxiliary learning compared to single task learning and multitask learning

Let’s look at a striking example [(Caruana et al., 1996)](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#caruana1996using) based on the Medis Pneumonia Database. This database contains 14,199 cases of patients diagnosed with pneumonia and hospitalized. It includes:

30 basic measurements (e.g. pulse);

35 lab results (e.g. blood counts);

the indication whether each patient survived or died.

Rich Caruana and his colleagues’ primary task was to identify patients that were more likely to survive and should then not be hospitalized. In this context, the lab results, available only after hospitalization, couldn’t be used as inputs. However, it was still possible to take advantage of them. Rich Caruana and his colleagues tried, as an auxiliary task, to predict the lab results based on the basic measurements. They thus reduced the error rate for their primary task by 5-10%.

Auxiliary learning has often been successfully applied in other settings, such as in the examples below:

|  |  |  |  |
| --- | --- | --- | --- |
| Domain | Primary task | Auxiliary tasks | Source |
| Deep learning | Training a deep neural network | Reconstructing corrupted versions of the neural network’s inputs and the activations of its hidden layers to learn useful features | [(Vincent et al., 2008)](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#vincent2008extracting) |
| Computer vision | Identifying facial landmarks on face pictures | Estimating head pose and predicting facial attributes (“wearing glasses”, “smiling” and “gender”) | [(Zhang et al., 2014)](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#Zhang14faciallandmark) |
| Computer vision | Detecting objects in indoor scenes | Predicting scene labels and evaluating depth and surface orientation at pixel level | [(Mordan et al., 2018)](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#mordan2018revisiting) |
| Sequence modelling | Training a recurrent neural network on very long sequences | Reconstructing or predicting short segments of the sequences | [(Trinh et al., 2018)](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#trinh2018learning) |
| Natural language processing | Performing a natural language processing task | Predicting words based on their neighborhood to learn efficient word representations | [(Mikolov et al., 2013)](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#mikolov2013distributed) |
| Reinforcement learning | Playing a video game | Modifying the image perceived by the agent and predicting short-term rewards | [(Jaderberg et al., 2016)](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#jaderberg2016reinforcement) |
| Reinforcement learning | Playing a video game | Predicting the future state based on the current state and the current action | [(Burda et al., 2018)](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#burda2018large) |

Table 2. Auxiliary learning has been successfully used in a variety of settings

Standard approach for deep auxiliary learning

For the sake of clarity, let’s consider only the case of a single primary task and a single auxiliary task (generalizing to an arbitrary number of primary and auxiliary tasks is straightforward). Let’s suppose further that these tasks correspond to the minimization of a primary loss LprimaryLprimary and an auxiliary loss LauxiliaryLauxiliary.

In the context of deep learning, the standard approach[2](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#fn:2) is to use a single neural network for both tasks, with shared layers followed by task-specific layers, and to apply a gradient descent-based method to minimize the weighted sum of the two losses L=Lprimary+λLauxiliaryL=Lprimary+λLauxiliary.

The corresponding gradient is ∇L=∇Lprimary+λ∇Lauxiliary∇L=∇Lprimary+λ∇Lauxiliary.

The underlying intuition is that minimizing LL will lead to more meaningful representations in the shared layers and that these representations will be leveraged by the layers specific to the primary task.

In the standard multitask approach, λλ is a constant value[3](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#fn:3). Its purpose is to balance the primary task and the auxiliary task. λ=0λ=0 corresponds to single task learning whereas increasing λλ gives more and more influence to the auxiliary task.

An alternative approach

The approach described above has provided very good empirical results. However, an auxiliary task may be unhelpful or even harmful for the primary task. The latter case, which is called negative transfer, is a central challenge of auxiliary learning.

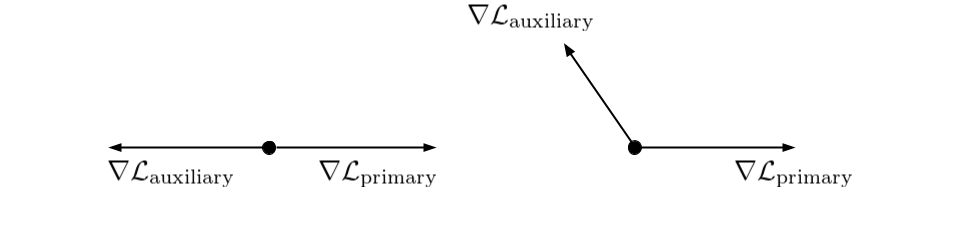


Figure 1. Left: a pathological example of negative transfer with an auxiliary task chosen so that Lauxiliary=−LprimaryLauxiliary=−Lprimary. In this case, ∇L=0∇L=0 (assuming λ=1λ=1) and the network's parameters are never updated during gradient descent. Right: a less extreme situation in which ∇Lauxiliary∇Lauxiliary still cancels a significant part of ∇Lprimary∇Lprimary

Moreover, using a weighted average of LprimaryLprimary and LauxiliaryLauxiliary suggests we’d be willing to tolerate a higher value of LprimaryLprimary if it helped sufficiently decrease LauxiliaryLauxiliary. On the contrary, reducing the auxiliary loss is only looked for to the extent this contributes to reducing the primary loss.

To mitigate negative transfer and better reflect the fundamental asymmetry between the primary loss and the auxiliary loss, I suggest substituting ∇L∇L with:

G=∇Lprimary+λGauxiliaryG=∇Lprimary+λGauxiliary

where GauxiliaryGauxiliary is:

as close as possible to ∇Lauxiliary∇Lauxiliary, to preserve the insights brought by the auxiliary task;

closer to ∇Lprimary∇Lprimary than −∇Lprimary−∇Lprimary, to avoid following a direction detrimental to the primary task.

Otherwise said, GauxiliaryGauxiliary is the projection of ∇Lauxiliary∇Lauxiliary on the half-space of vectors whose cosine similarity with ∇Lprimary∇Lprimary is positive. If a critical point of LprimaryLprimary hasn’t been reached yet (i.e. if ∇Lprimary≠0∇Lprimary≠0), this ensures that LprimaryLprimary will decrease if the learning rate if small enough.



Figure 2. If ∇Lprimary∇Lprimary and ∇Lauxiliary∇Lauxiliary form an obtuse angle, the component of ∇Lauxiliary∇Lauxiliary parallel to ∇Lprimary∇Lprimary is cancelled to avoid increasing LprimaryLprimary. If not, ∇Lauxiliary∇Lauxiliary is kept as it is

Comparison to unweighted and weighted cosine

This approach, referred below as projection, is analogous to two methods, unweighted cosine and weighted cosine, recently proposed for the same purpose [(Du et al., 2018)](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#du2018adapting). Yunshu Du, Wojciech Czarnecki and their colleagues also suggest to adjust ∇Lauxiliary∇Lauxiliary:

|  |  |
| --- | --- |
| Method | Adjusted auxiliary loss gradient GauxiliaryGauxiliary |
| Unweighted cosine | {∇Lauxiliaryif cos(∇Lprimary,∇Lauxiliary)≥00if cos(∇Lprimary,∇Lauxiliary)<0{∇Lauxiliaryif cos⁡(∇Lprimary,∇Lauxiliary)≥00if cos⁡(∇Lprimary,∇Lauxiliary)<0 |
| Weighted cosine | max(0,cos(∇Lprimary,∇Lauxiliary)).∇Lauxiliarymax(0,cos⁡(∇Lprimary,∇Lauxiliary)).∇Lauxiliary |
| Projection | ∇Lauxiliary−min(0,∇Lauxiliary.∇Lprimary||∇Lprimary||).∇Lprimary||∇Lprimary||∇Lauxiliary−min(0,∇Lauxiliary.∇Lprimary||∇Lprimary||).∇Lprimary||∇Lprimary|| |

Table 3. Unweighted cosine and weighted cosine compared to the method proposed here

If ∇Lprimary∇Lprimary and ∇Lauxiliary∇Lauxiliary form an acute angle, both projection and unweighted cosine leave ∇Lauxiliary∇Lauxiliary unchanged whereas weighted cosine reduces its norm with the cosine similarity of these vectors. In particular, weighted cosine strongly shrinks ∇Lauxiliary∇Lauxiliary when it becomes almost orthogonal to ∇Lprimary∇Lprimary.

If ∇Lprimary∇Lprimary and ∇Lauxiliary∇Lauxiliary form an obtuse angle, both unweighted cosine and weighted cosine just ignore ∇Lauxiliary∇Lauxiliary. In contrast, projection only negates the component of ∇Lauxiliary∇Lauxiliary which is collinear to ∇Lprimary∇Lprimary.

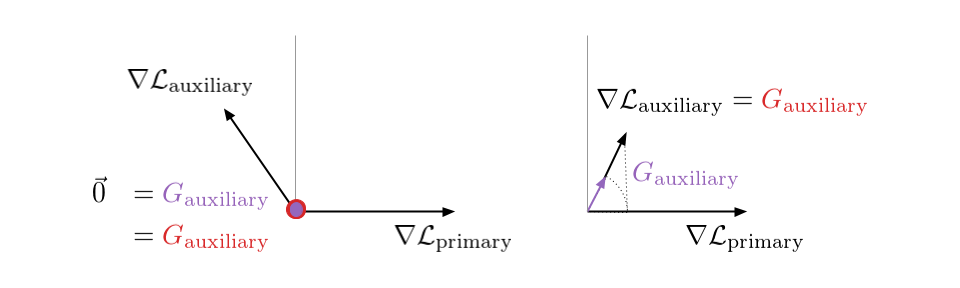


Figure 3. ∇Lauxiliary∇Lauxiliary transformed into GauxiliaryGauxiliary by unweighted cosine (red) and weighted cosine (purple)

Even if projection and unweighted/weighted cosine are similar, their motivations are different:

[(Du et al., 2018)](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#du2018adapting) explicitly tries to favor auxiliary tasks similar to the primary task. For example, weighted cosine completely ignores the auxiliary task when ∇Lauxiliary∇Lauxiliary and ∇Lprimary∇Lprimary are orthogonal.

In my intuition, an auxiliary task’s purpose is to bring a new perspective. It should nudge us towards interesting regions of the parameter space we wouldn’t have explored with only the primary task in focus. Therefore we can try to negate the component of ∇Lauxiliary∇Lauxiliary which seems clearly harmful to our objective but we shouldn’t necessarily discard auxiliary tasks dissimilar to the primary task[4](https://vivien000.github.io/blog/journal/learning-though-auxiliary_tasks.html#fn:5).

Experiments

Let’s now look at some experiments. They were performed with Tensorflow and you can find the corresponding notebooks and additional results on [GitHub](https://github.com/vivien000/auxiliary-learning).

Toy experiments

We start with a minimalistic example to illustrate how substituting ∇Lauxiliary∇Lauxiliary with GauxiliaryGauxiliary gives priority to the primary task while still allowing progress on the auxiliary task.

In the euclidean plane R2R2, we try to find a point (x,y)(x,y) that, above all, minimizes the square distance to the unit circle and, only if it doesn’t increase this distance, also minimizes the square distance to point (2, 0).

Otherwise said, Lprimary=(√x2+y2−1)2Lprimary=(x2+y2−1)2 and Lauxiliary=(x−2)2+y2Lauxiliary=(x−2)2+y2.

This is a slight departure from the principle of auxiliary learning described above. Here we want to accomplish both the primary task and the auxiliary task even if the primary task is the absolute priority (performing the primary task alone would be too easy).

The solution of this problem is of course (1,0)(1,0) and we try to reach it by using a gradient descent-based algorithm starting from point (0,2)(0,2) with λ=0.1λ=0.1.

We can see the corresponding trajectories with the Adam algorithm (learning rate: 0.01, β1=0.9,β2=0.999β1=0.9,β2=0.999) on Figure 4 and observe that:

With multitask, the point converges to a location close to but different from (1, 0) because of the distraction the auxiliary task creates.

With projection, the point follows as expected the same trajectory as with multitask before reaching the disk of radius 1 and center (1, 0), i.e. as long as ∇Lprimary∇Lprimary and ∇Lauxiliary∇Lauxiliary form an acute angle. Afterwards, the point slightly deviates from the multitask trajectory to reach (1, 0).

The trajectory for unweighted cosine is also the same before reaching the disk of radius 1 and center (1, 0). Afterwards, the point abruptly changes course in the direction of the unit disk as the auxiliary task is ignored. When the point arrives at the unit disk, it starts oscillating around its edge. It’s then intermittently influenced by the auxiliary task and follows the unit circle until reaching (1, 0).

With weighted cosine, the point follows a more direct trajectory towards the unit circle. It then arrives close to the intersection between the unit circle and the circle of radius 1 and center (1, 0), i.e. a location where ∇Lprimary∇Lprimary and ∇Lauxiliary∇Lauxiliary are orthogonal. It stagnates there for a while because the auxiliary task is mostly ignored then converges to (1, 0) by following the unit circle.

With this example, projection, unweighted cosine and weighted cosine allow to find the correct solution but the convergence is notably quicker with projection (this is also the case for [other](https://vivien000.github.io/blog/assets/img/trajectories_adam2.gif) [starting](https://vivien000.github.io/blog/assets/img/trajectories_adam3.gif) [points](https://vivien000.github.io/blog/assets/img/trajectories_adam4.gif)).

 With unweighted cosine and weighted cosine, the point now stops on the unit circle far from (1, 0). This is because, in the absence of momentum, it eventually stays on the side of the unit circle where ∇Lprimary∇Lprimary and ∇Lauxiliary∇Lauxiliary form an obtuse angle. The auxiliary task is then never taken into consideration anymore. A similar behavior can be seen with [other](https://vivien000.github.io/blog/assets/img/trajectories_vanilla_gd2.gif) [starting](https://vivien000.github.io/blog/assets/img/trajectories_vanilla_gd3.gif) [points](https://vivien000.github.io/blog/assets/img/trajectories_vanilla_gd4.gif).

In summary, the proposed method works as intended with these toy examples: it helps accomplish both the primary task and the auxiliary task while giving precedence to the former.