**Batch Normalization in practice: an example with Keras and TensorFlow 2.0**

A step by step tutorial to add and customize batch normalization

In this article, we will focus on adding and customizing batch normalization in our machine learning model and look at an example of how we do this in practice with Keras and TensorFlow 2.0.

**A gentle introduction to batch normalization**

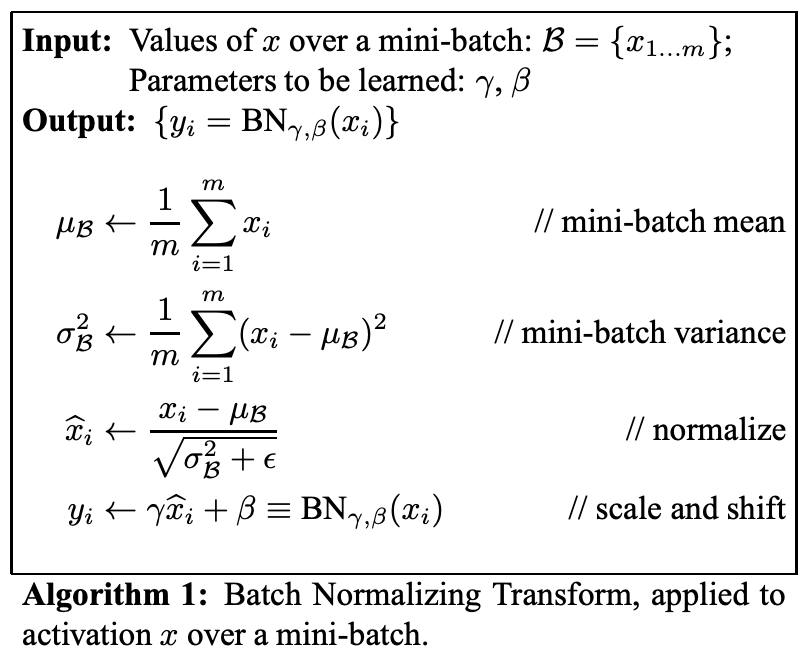
In the rise of deep learning, one of the most important ideas has been an algorithm called ***batch normalization***(also known as ***batch norm***).

***Batch normalization*** is a technique for training very deep neural networks that standardizes the inputs to a layer for each mini-batch. This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks.

By [Jason Brownlee](https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/)

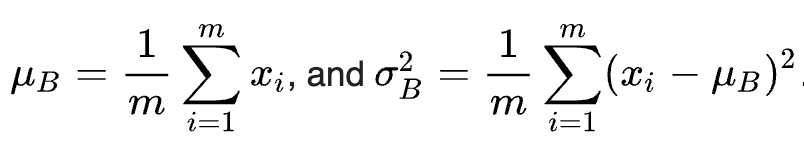
Batch normalization can be implemented during training by calculating the mean and standard deviation of each input variable to a layer per mini-batch and using these statistics to perform the standardization.

Formally, the batch normalization algorithm [1] is defined as:



From the original paper: [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](https://arxiv.org/pdf/1502.03167v3.pdf)

In the algorithm, *B* is used to denote a mini-batch of size m of the entire training set. The mean and variance of *B* could thus be calculated as:

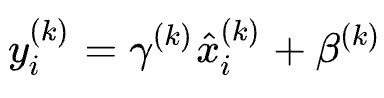


For a layer with *d*-dimensional input, *x* = (*x\_1*, …, *x\_d*), each dimension of its input can be normalized (re-centered and rescaled) separately. Thus, the normalization for a *d*-dimensional input can be calculated as:



ε is added in the denominator for numerical stability and is an arbitrarily small constant.

And finally, to restore the representation power of the network, a transformation step is defined as:



where the parameter β and γ are subsequently learned in the optimization process.

The benefits of batch normalization are [2]:

* **A deep neural network can be trained faster:** Although each training iteration will be slower because of the extra normalization calculation during the forward pass and the additional hyperparameters to train during backpropagation, it should converge much more quickly; thus, training should be faster overall.
* **Higher learning rate:**Gradient descent generally requires small learning rates for the network to converge. As networks become deeper, gradients become smaller during backpropagation and thus require even more iterations. Using batch normalization allows much higher learning rates, thereby increasing the speed of training.
* **Easier to initialize weight:** Weight initialization can be difficult, particularly when creating deeper networks. Batch normalization reduces the sensitivity to the initial starting weights.

If you are looking for a complete explanation, you might find the following resources useful:

* [The original paper](https://arxiv.org/abs/1502.03167)
* [Batch Normalization in Deeplearning.ai](https://www.coursera.org/lecture/deep-neural-network/normalizing-activations-in-a-network-4ptp2)

In the following article, we are going to add and customize batch normalization in our machine learning model.

**Environment setup, Source code, and dataset preparation**

We will be using the same dataset as we did in the [model regularization tutorial](https://towardsdatascience.com/machine-learning-model-regularization-in-practice-an-example-with-keras-and-tensorflow-2-0-52a96746123e). You can skip this chapter if you are already familiar with it.

In order to run this tutorial, you need to install

*TensorFlow 2, numpy, pandas, sklean, matplotlib*

They can all be installed directly vis PyPI and I strongly recommend to create a new Virtual Environment. For a tutorial on creating a Python virtual environment

* [Create Virtual Environment using “virtualenv” and add it to Jupyter Notebook](https://towardsdatascience.com/create-virtual-environment-using-virtualenv-and-add-it-to-jupyter-notebook-6e1bf4e03415)
* [Create Virtual Environment using “conda” and add it to Jupyter Notebook](https://medium.com/analytics-vidhya/create-virtual-environment-using-conda-and-add-it-to-jupyter-notebook-d319a81dfd1)

**Source code**

This is a step by step tutorial and all instructions are in this article. For source code, please check out my Github [machine learning repo](https://github.com/BindiChen/machine-learning).

**Dataset preparation**

This tutorial uses the [Anderson Iris flower (iris)](https://en.wikipedia.org/wiki/Iris_flower_data_set) dataset for demonstration. The dataset contains a set of 150 records under five attributes: *sepal length*, *sepal width*, *petal length*, *petal width,* and *class* (known as *target* from sklearn datasets).

First, let’s import the libraries and obtain iris dataset from ***scikit-learn*** library. You can also download it from the [UCI Iris dataset](https://archive.ics.uci.edu/ml/datasets/iris).

import tensorflow as tf  
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from sklearn.datasets import load\_iris  
from sklearn.model\_selection import train\_test\_split**iris = load\_iris()**

For the purpose of exploring data, let’s load data into a DataFrame

# Load data into a DataFrame  
**df = pd.DataFrame(iris.data, columns=iris.feature\_names)**# Convert datatype to float  
**df = df.astype(float)**# append "target" and name it "label"  
**df['label'] = iris.target**# Use string label instead  
**df['label'] = df.label.replace(dict(enumerate(iris.target\_names)))**

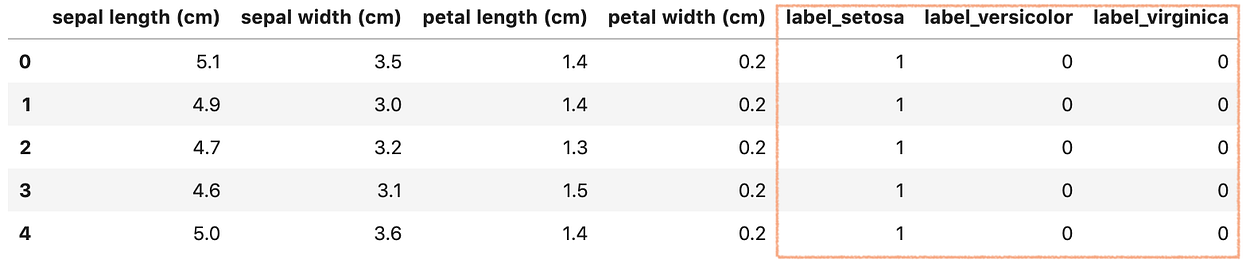
And the df should look like below:



We notice the ***label*** column is a categorical feature and will need to convert it to [one-hot encoding](https://towardsdatascience.com/what-is-one-hot-encoding-and-how-to-use-pandas-get-dummies-function-922eb9bd4970). Otherwise, our machine learning algorithm won’t be able to directly take in that as input.

# label -> one-hot encoding  
**label = pd.get\_dummies(df['label'], prefix='label')**  
**df = pd.concat([df, label], axis=1)**  
# drop old label  
df.drop(['label'], axis=1, inplace=True)

Now, the df should look like:



Next, let’s create X and y. Keras and TensorFlow 2.0 only take in Numpy array as inputs, so we will have to convert DataFrame back to Numpy array.

# Creating X and y**X = df[['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']]**  
# Convert DataFrame into np array  
**X = np.asarray(X)y = df[['label\_setosa', 'label\_versicolor', 'label\_virginica']]**# Convert DataFrame into np array  
**y = np.asarray(y)**

Finally, let’s split the dataset into a training set (80%)and a test set (20%) using **train\_test\_split()**from **sklearn** library.

X\_train, X\_test, y\_train, y\_test = **train\_test\_split**(  
 **X,  
 y,  
 test\_size=0.20**  
)

Great! our data is ready for building a Machine Learning model.

**Build a neural network model with batch normalization**

There are [3 ways to create a machine learning model with Keras and TensorFlow 2.0](https://towardsdatascience.com/3-ways-to-create-a-machine-learning-model-with-keras-and-tensorflow-2-0-de09323af4d3). Since we are building a simple fully connected neural network and for simplicity, let’s use the easiest way: Sequential Model with Sequential().

First, let’s import Sequential and BatchNormalization

from tensorflow.keras.models import **Sequential**  
from tensorflow.keras.layers import Dense, **BatchNormalization**

Let’s go ahead and create a sequential model

model = Sequential([  
 Dense(64, **input\_shape=(4,)**, activation="relu"),  
 Dense(128, activation='relu'),  
 Dense(128, activation='relu'),  
 Dense(64, activation='relu'),  
 Dense(64, activation='relu'),  
 Dense(3, **activation='softmax'**)  
]);

Our model has the following specifications:

* The first layer (also known as the input layer) has the input\_shape to set the input size (4,)
* The input layer has 64 units, followed by 2 dense layers, each with 128 units. Then there are further 2dense layers, each with 64 units. All these layers use the relu activation function.
* The output Dense layer has 3 units and the softmax activation function.

We can add batch normalization into our model by adding it in the same way as adding Dense layer.

model = Sequential([  
 Dense(64, input\_shape=(4,), activation="relu"),  
 **BatchNormalization(),**  
 Dense(128, activation='relu'),  
 **BatchNormalization(),**  
 Dense(128, activation='relu'),  
 **BatchNormalization(),**  
 Dense(64, activation='relu'),  
 **BatchNormalization(),**  
 Dense(64, activation='relu'),  
 **BatchNormalization(),**  
 Dense(3, activation='softmax')  
]);

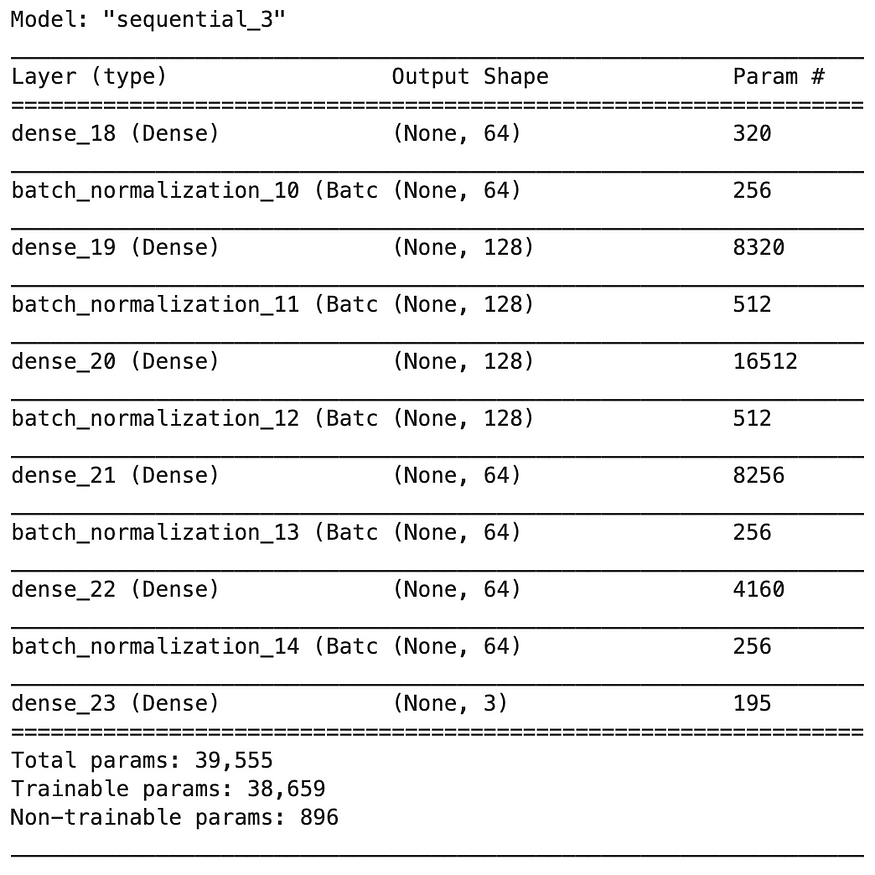
BatchNormalization() normalize the activation of the previous layer at each batch and by default, it is using the following values [3]:

* Momentum defaults to 0.99
* The hyperparameter ε defaults to 0.001
* The hyperparameter β defaults to an all-zeros vector
* The hyperparameter γ defaults to an all-ones vector

These can all be changed by adding optional arguments to BatchNormalization() . For example

from tensorflow.keras.initializers import **RandomNormal, Constant**# Model with default batch normalization  
model = Sequential([  
 Dense(64, input\_shape=(4,), activation="relu"),  
 BatchNormalization(),  
 Dense(128, activation='relu'),  
 BatchNormalization(),  
 Dense(128, activation='relu'),  
 BatchNormalization(),  
 Dense(64, activation='relu'),  
 BatchNormalization(),  
 Dense(64, activation='relu'),  
 **BatchNormalization(  
 momentum=0.95,   
 epsilon=0.005,  
 beta\_initializer=RandomNormal(mean=0.0, stddev=0.05),   
 gamma\_initializer=Constant(value=0.9)  
 ),**  
 Dense(3, activation='softmax')  
]);

RandomNormal() generates a tensor with a normal distribution and Constant() generates a tensor with constant values. By running model.summary() you should get a model summary like below:



**Training**

Let’s now compile and fit our model with batch normalization. We first compile our model with the following specifications

* Use Adam (adam) optimization algorithm as the optimizer
* Use categorical cross-entropy loss function (categorical\_crossentropy) for our ***multiple-class classification*** problem
* For simplicity, use accuracy as our evaluation metrics to evaluate the model during training and testing.

model.compile(  
 **optimizer='adam',   
 loss='categorical\_crossentropy',   
 metrics=['accuracy']**  
)

After that, we can call model.fit() to fit our model to the training data.

history = model.fit(  
 X\_train,   
 y\_train,   
 epochs=200,   
 validation\_split=0.25,   
 batch\_size=40,   
 verbose=2  
)

If all runs smoothly, we should get an output like below

Train on 90 samples, validate on 30 samples  
Epoch 1/200  
90/90 - 3s - loss: 0.8735 - accuracy: 0.5778 - val\_loss: 1.0685 - val\_accuracy: 0.6667  
Epoch 2/200  
90/90 - 0s - loss: 0.1983 - accuracy: 0.9333 - val\_loss: 1.0640 - val\_accuracy: 0.4667  
......  
......  
Epoch 200/200  
90/90 - 0s - loss: 0.0532 - accuracy: 0.9778 - val\_loss: 0.1453 - val\_accuracy: 0.9333

**Model Evaluation**

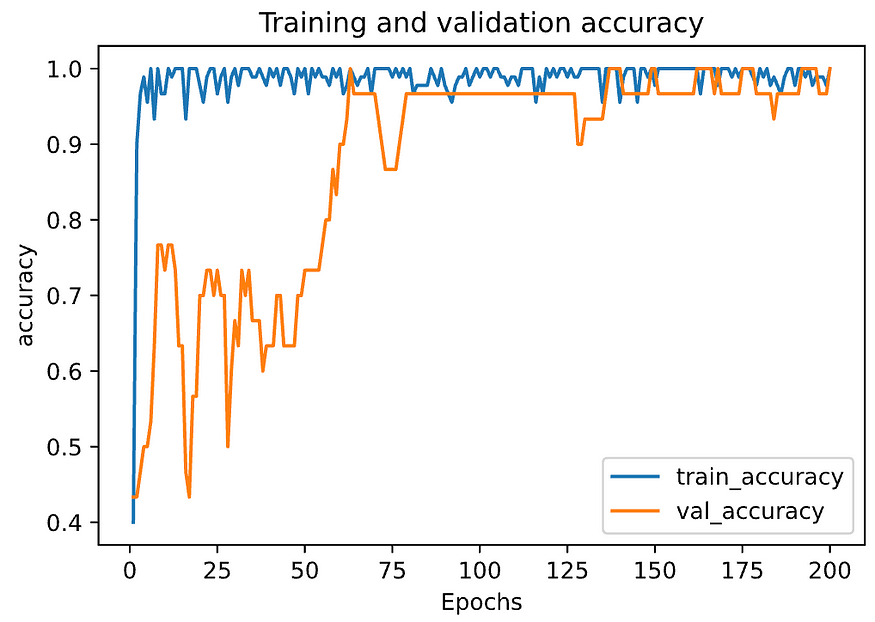
Finally, it’s time to see if the model is any good by

* Plotting training and validation loss and accuracy to observe how the accuracy of our model improves over time.
* Test our model again the test dataset X\_test that we set aside earlier

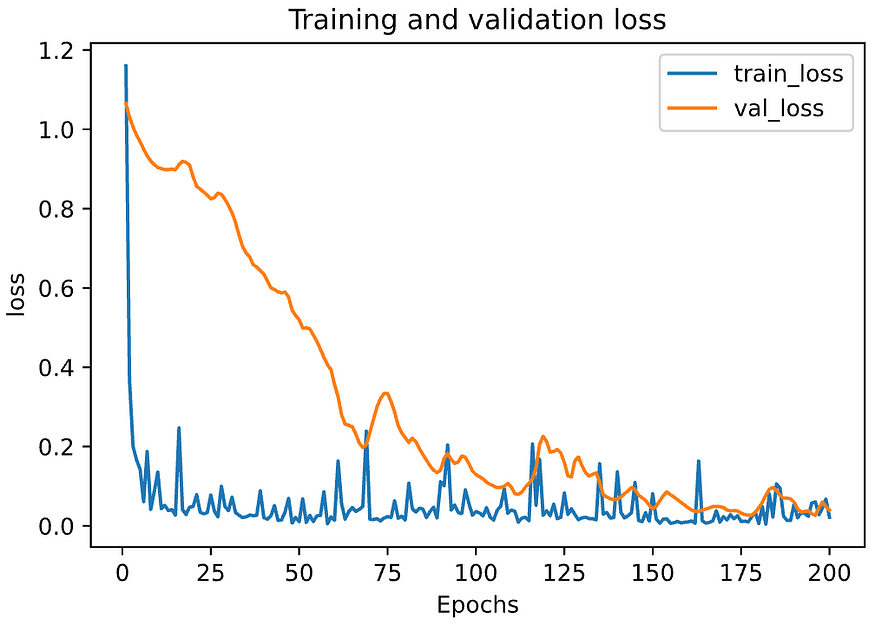
Let’s create a function plot\_metric() for plotting metrics.

%matplotlib inline  
%config InlineBackend.figure\_format = 'svg'  
def plot\_metric(history, metric):  
 train\_metrics = history.history[metric]  
 val\_metrics = history.history['val\_'+metric]  
 epochs = range(1, len(train\_metrics) + 1)  
 plt.plot(epochs, train\_metrics)  
 plt.plot(epochs, val\_metrics)  
 plt.title('Training and validation '+ metric)  
 plt.xlabel("Epochs")  
 plt.ylabel(metric)  
 plt.legend(["train\_"+metric, 'val\_'+metric])  
 plt.show()

By running plot\_metric(history, 'accuracy') to plot the progress on accuracy.



By running plot\_metric(history, 'loss') to plot the progress on loss.



To evaluate the model on the test set

# Evaluate the model on the test set  
model.**evaluate**(**X\_test**, **y\_test**, verbose=2)

And we should get an output like below

30/1 - 0s - loss: 0.1192 - accuracy: 0.9667  
[0.11924469470977783, 0.96666664]