BTS | Barcelona Technology School

Real Time Data Analysis

Master in Big Data Solutions 2020-2021

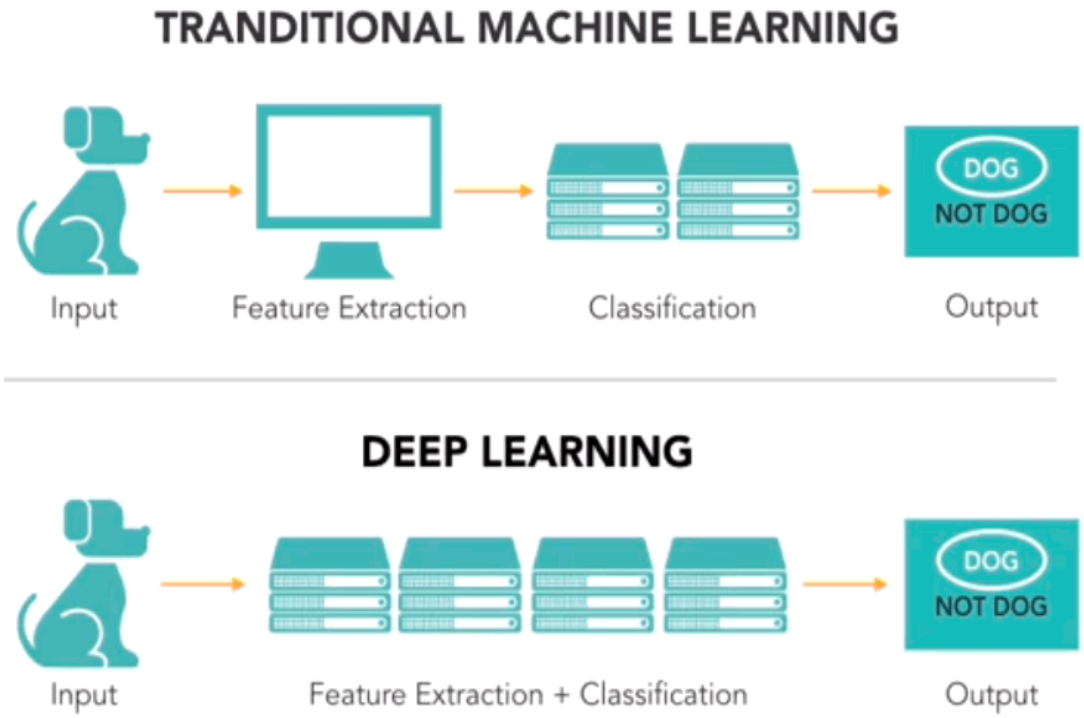
Contents

What's in the menu today?

- Machine Learning vs Deep Learning
- Introduction to Deep Learning in Keras
- Learning more about creating models in Keras
- Distributed Deep Learning model training
- Introduction to Distributed Deep Learning in Elephas

Introduction to Deep Learning

Machine Learning vs Deep Learning



Introduction to Deep Learning

Machine Learning vs Deep Learning

MACHINE LEARNING	DEEP LEARNING
Optimal data volumes	
Thousands of data points	Big data: millions of data points
Outputs	
Numerical value, like classification or score	Anything from numerical values to free-form elements, such as free text and sounds
How it works	
Uses various types of automated algorithms that learn to model functions and predict future actions from data	Uses neural networks that pass data through many processing layers to interpret data features and relations
How it's managed	
Algorithms are detected by data analysts to examine specific variables in data sets	Algorithms are largely self-directed on data analysis once they're put into production

Introduction to Deep Learning

What is Keras?

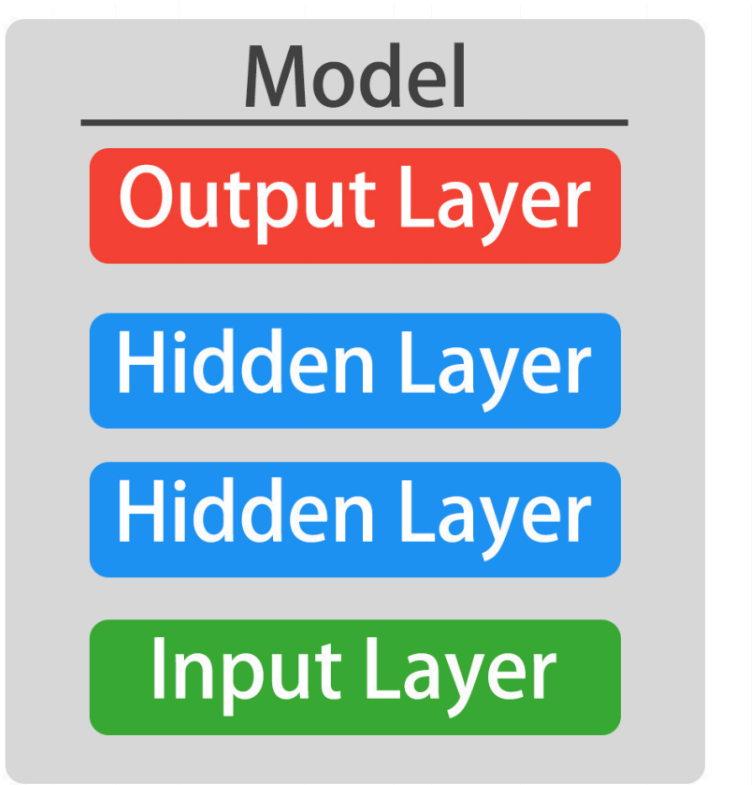
What is Keras?
- Deep Learning Framework
- Enables fast experimentation
- Runs on top of other frameworks
- Authored by François Chollet

Why Keras is Preferred?
- Lesser code
- Faster prototyping
- Easy to build and modify different model architectures
- Deployment in multiple platforms

Introduction to Deep Learning

Deep Learning Modelling in Keras

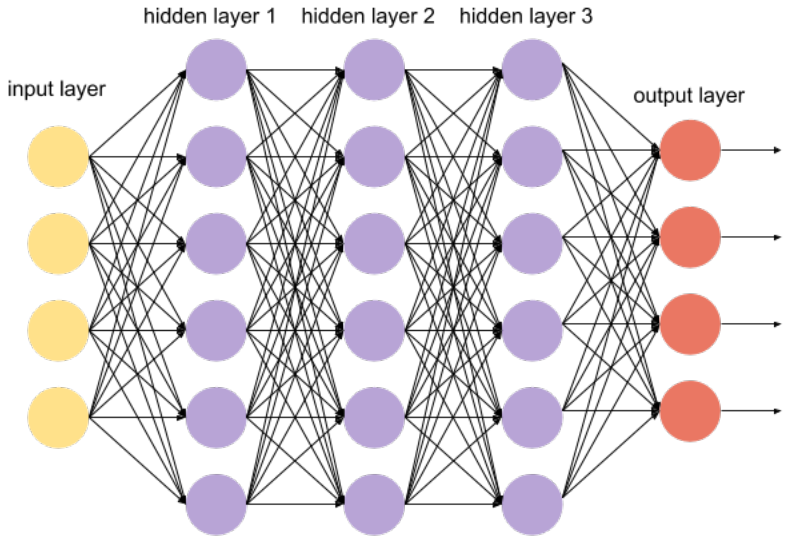
In **Keras**, ideally, the models are built in terms of layers which are placed successively over the previous layers. This sequential modelling offers simplicity to expand a model in terms of variety of layers, functions operating on those layers etc.



Introduction to Deep Learning

Fully Connected Dense Neural Network

In **Keras**, ideally, the models are built in terms of layers which are placed successively over the previous layers. This sequential modelling offers simplicity to expand a model in terms of variety of layers, functions operating on those layers etc.



Observe that the various layers of the model are placed one after the other in a sequence. Right after the input layer comes the firs hidden layer followed by the second and third and so on so forth..

Introduction to Deep Learning

Reading and Transforming Data

Ensure that both of your training and testing datasets are stored in a folder called “data_path”. We are using a custom function written by us to read the data. The same must be checked and accordingly modified in the jupyter-notebook.

```
%%capture
data_path= "../data_path/"
train, test= extracting_training_test_data(data_path)
```

```
array(['Class_2', 'Class_8', 'Class_6', 'Class_3', 'Class_7', 'Class_4',
       'Class_5', 'Class_9', 'Class_1'], dtype=object)
```

After reading the data, we are modifying the class labels in order to make them numeric of int type. It was achieved without the use of any pre-processing library. Although you’re free to check the functions offered by Keras regarding pre-processing.

```
# it's easy enough to encode the classes without an extra module
train['target'] = [x[-1] for x in train['target']]
train['target'] = train['target'].astype(int) - 1
train.target.unique()
```

```
array([1, 7, 5, 2, 6, 3, 4, 8, 0])
```

Introduction to Deep Learning

Splitting and Normalising the Data

After modifying the class labels, we are splitting the data into appropriate training and testing sets as well as training and testing labels using a pre-determined test_size.

```
from sklearn.model_selection import train_test_split
# slice features and classes from df
X = train.iloc[:, 1:94].values.astype(float)
y = train.iloc[:, 94].values.astype(float)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)
```

The dataset was standardised by means of a standard scaler before serving the data into the model. It is usually an essential step to perform such scaling.

```
from sklearn.preprocessing import StandardScaler

# scale features
sc = StandardScaler()
X = sc.fit_transform(X)
X_train = sc.fit_transform(X_train)
X_test = sc.fit_transform(X_test)
```

Major advantages of scaling the data-

- It is observed that normalizing the data generally speeds up learning and leads to faster convergence
- It allows the features to be comparable when the model will be attempting to learn from them

Introduction to Deep Learning

Building the first Keras Model

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

def classifier():
    classifier = Sequential()
    classifier.add(Dense(units = 200, input_dim = 93))
    classifier.add(Dense(units = 50))
    classifier.add(Dense(units = 9, activation = 'softmax'))
    classifier.compile(optimizer = 'sgd',
                       loss = 'sparse_categorical_crossentropy', metrics = ['accuracy'])
    return classifier
classifier = classifier()
history = classifier.fit(X_train, y_train,
                         validation_data=(X_test,y_test),
                         epochs=10, batch_size=1000)
```

```
Epoch 1/10
50/50 [==============================] - 1s 6ms/step - loss: 2.0497 - accuracy: 0.3643 - val_loss: 1.2157 - val_acc
uracy: 0.6320
Epoch 2/10
50/50 [==============================] - 0s 3ms/step - loss: 1.1321 - accuracy: 0.6570 - val_loss: 1.0171 - val_acc
uracy: 0.6889
Epoch 3/10
50/50 [==============================] - 0s 3ms/step - loss: 0.9755 - accuracy: 0.6971 - val_loss: 0.9334 - val_acc
uracy: 0.7016
Epoch 4/10
50/50 [==============================] - 0s 3ms/step - loss: 0.9030 - accuracy: 0.7084 - val_loss: 0.8836 - val_acc
uracy: 0.7084
Epoch 5/10
50/50 [==============================] - 0s 3ms/step - loss: 0.8605 - accuracy: 0.7171 - val_loss: 0.8496 - val_acc
uracy: 0.7149
Epoch 6/10
50/50 [==============================] - 0s 3ms/step - loss: 0.8276 - accuracy: 0.7238 - val_loss: 0.8251 - val_acc
uracy: 0.7184
Epoch 7/10
50/50 [==============================] - 0s 3ms/step - loss: 0.8066 - accuracy: 0.7238 - val_loss: 0.8060 - val_acc
uracy: 0.7214
Epoch 8/10
50/50 [==============================] - 0s 3ms/step - loss: 0.7917 - accuracy: 0.7282 - val_loss: 0.7906 - val_acc
uracy: 0.7249
Epoch 9/10
50/50 [==============================] - 0s 3ms/step - loss: 0.7777 - accuracy: 0.7310 - val_loss: 0.7781 - val_acc
uracy: 0.7275
Epoch 10/10
50/50 [==============================] - 0s 3ms/step - loss: 0.7632 - accuracy: 0.7333 - val_loss: 0.7682 - val_acc
uracy: 0.7279
```

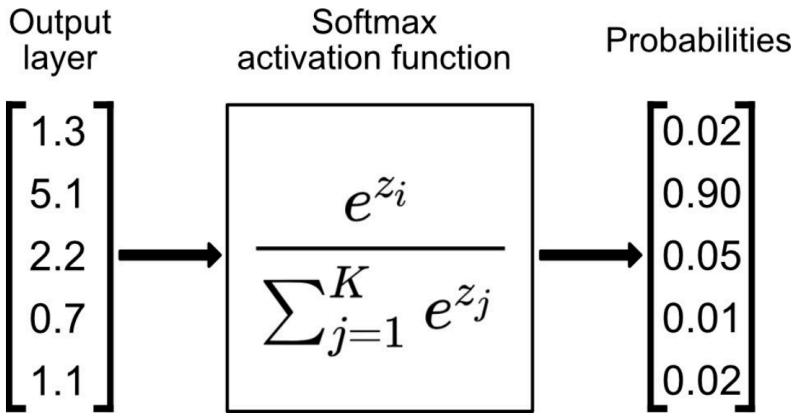
Introduction to Deep Learning

Activation Function: SoftMax

The mathematical formula for the SoftMax function is defined as follows-

$$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

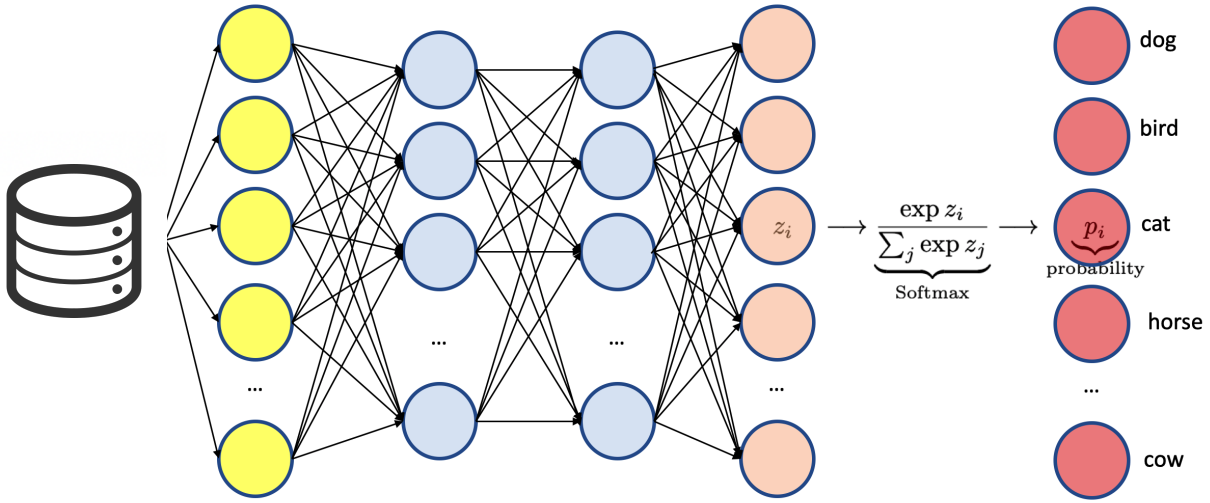
Basically, the activation function (SoftMax in this case) extracts the raw inputs from the final layer and applies a mathematical operation to transform them into probabilities.



It is interesting to see that the sum of all these probabilities is one which justifies the need of such a function.

Introduction to Deep Learning

Activation Function: SoftMax



Introduction to Deep Learning

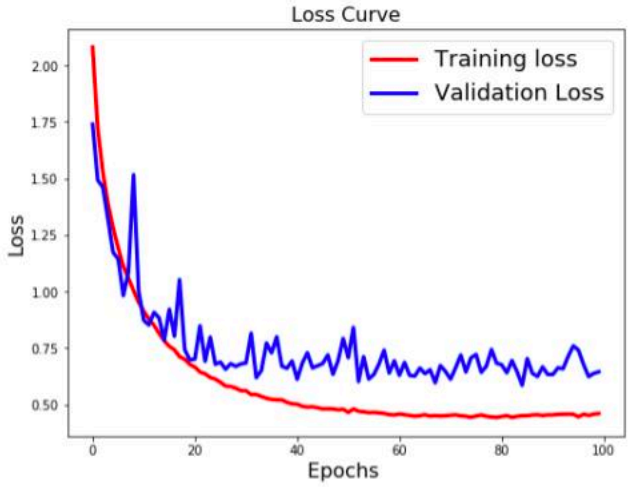
Investigating the model: Learning Curves

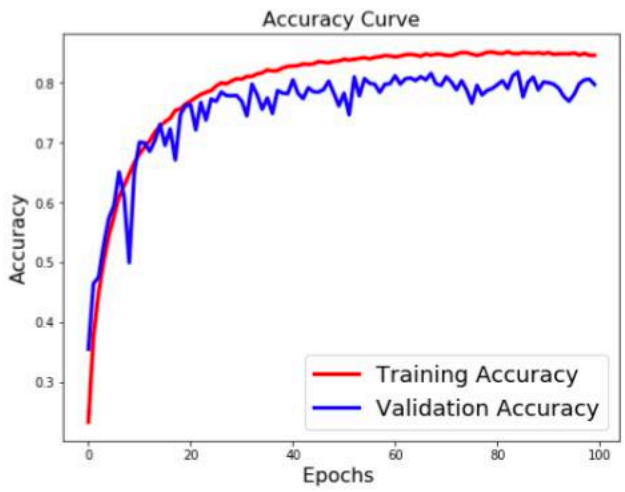
Typically, a Keras model provides us with two kinds of learning curves-

- Loss curve: describes the behaviour of our loss function as the model progresses to learn
- Accuracy curve: describes the behaviour of accuracy in predictions as the model progresses to learn

Ideally we expect that.

- Loss curve comes lower and lower as the epochs increase
- Accuracy curve goes higher and higher as the epochs increase





Introduction to Deep Learning

Investigating the model: Learning Curves

Typically, a Keras model provides us with two kinds of learning curves-

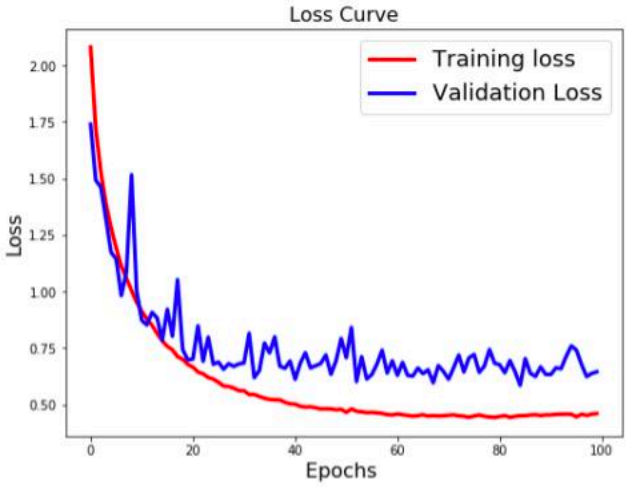
- Loss curve: describes the behaviour of our loss function as the model progresses to learn
- Accuracy curve: describes the behaviour of accuracy in predictions as the model progresses to learn

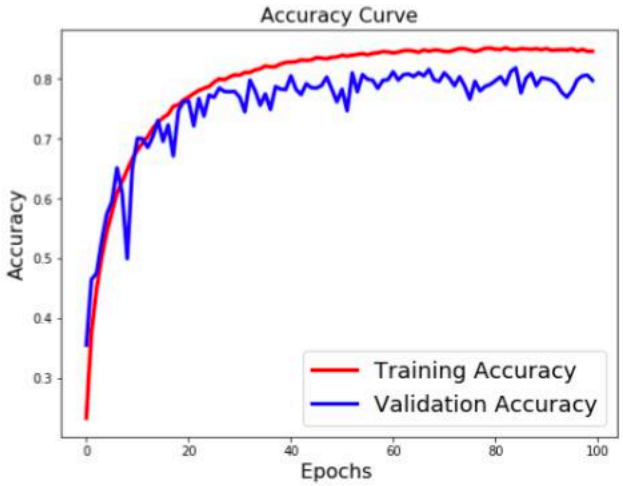
Ideally we expect that.

- Loss curve comes lower and lower as the epochs increase
- Accuracy curve goes higher and higher as the epochs increase

Note:

- Overfitting: When the difference between the training and validation loss as well as accuracy starts to increase significantly as the training progresses
- Adding more examples to learn: Accuracy will be generally expected to increase on both training and validation sets





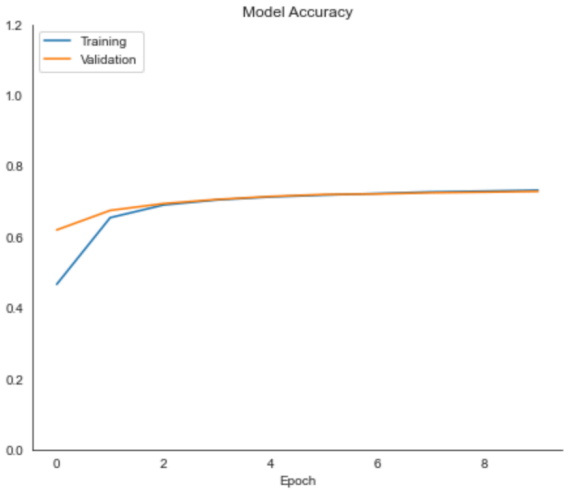
Introduction to Deep Learning

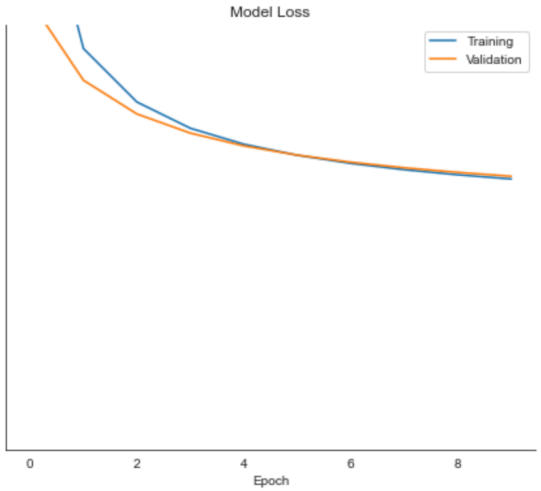
Investigating the model: Learning Curves

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns

def plots(acc, val_acc, loss, val_loss):
    fig, (ax1, ax2) = plt.subplots(ncols=2, sharey=True)

    # plot accuracy
    ax1.plot(acc, label='Training')
    if val_acc is not None:
        ax1.plot(val_acc, label='Validation')

    ax1.set_xlabel('Epoch')
    ax1.set_title('Model Accuracy')
    ax1.legend(loc='upper left')
    ax1.set_ylim([0,1.2])

    # plot loss
    ax2.plot(loss, label='Training')
    if val_loss is not None:
        ax2.plot(val_loss, label='Validation')

    ax2.set_xlabel('Epoch')
    ax2.set_title('Model Loss')
    ax2.legend(loc='upper right')
    ax2.set_ylim([0,1.2])

    sns.despine()
    sns.set_style("white")


    plt.show()

plt.rcParams["figure.figsize"] = (16,6)
plots(history.history["accuracy"],history.history['val_accuracy'],
      history.history["loss"],history.history['val_loss'])
```

Introduction to Deep Learning

Investigating the model: Learning Curves





Introduction to Deep Learning

Activations in Hidden Layers

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

def classifier():
    classifier = Sequential()
    classifier.add(Dense(units = 200,
                         activation = 'relu', input_dim = 93))
    classifier.add(Dense(units = 50,
                         activation = 'relu'))
    classifier.add(Dense(units = 9,
                         activation = 'softmax'))
    classifier.compile(optimizer = 'sgd',
                       loss = 'sparse_categorical_crossentropy', metrics = ['accuracy'])
    return classifier
```

```
Epoch 1/10
50/50 [==============================] - 1s 8ms/step - loss: 2.1986 - accuracy: 0.1898 - val_loss: 1.8962 - val_acc
uracy: 0.4340
Epoch 2/10
50/50 [==============================] - 0s 3ms/step - loss: 1.8295 - accuracy: 0.4693 - val_loss: 1.6575 - val_acc
uracy: 0.5227
Epoch 3/10
50/50 [==============================] - 0s 3ms/step - loss: 1.6094 - accuracy: 0.5386 - val_loss: 1.4864 - val_acc
uracy: 0.5672
Epoch 4/10
50/50 [==============================] - 0s 3ms/step - loss: 1.4529 - accuracy: 0.5795 - val_loss: 1.3526 - val_acc
uracy: 0.6075
Epoch 5/10
50/50 [==============================] - 0s 3ms/step - loss: 1.3232 - accuracy: 0.6167 - val_loss: 1.2458 - val_acc
uracy: 0.6403
```

The major advantage offered when we add activations in the hidden layers is that it allows to learn more complex functions than a network trained using a linear activation function.

Introduction to Deep Learning

The Kernel Trick of Neural Networks

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

def classifier():
    classifier = Sequential()
    classifier.add(Dense(units = 200, kernel_initializer = 'uniform',
                         activation = 'relu', input_dim = 93))
    classifier.add(Dense(units = 50, kernel_initializer = 'uniform',
                         activation = 'relu'))
    classifier.add(Dense(units = 9, kernel_initializer = 'uniform',
                         activation = 'softmax'))
    classifier.compile(optimizer = 'rmsprop',
                       loss = 'sparse_categorical_crossentropy', metrics = ['accuracy'])
    return classifier
```

The neural network needs to start with some weights and then iteratively update them to better values. The term Kernel Initializer is a fancy term for which statistical distribution or function to use for initialising the weights.

In case of statistical distribution, the library will generate numbers from that statistical distribution and use as starting weights. For example in the above code, uniform distribution will be used to initialise weights. You can use other functions (constants like 1s or 0s) and distributions (normal) too.

The term kernel is a carryover from other classical methods like SVM. The idea is to transform data in a given input space to another space where the transformation is achieved using kernel functions. We can think of neural network layers as non-linear maps doing these transformations, so the term kernels is used.

Read more about the weight initialisers here

Introduction to Deep Learning

The Kernel Trick of Neural Networks

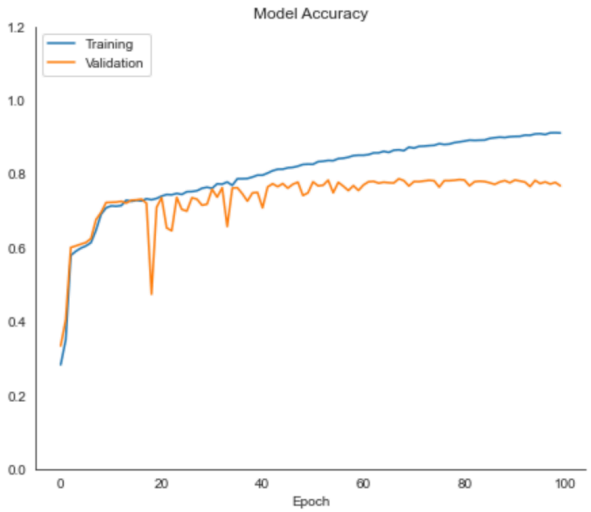
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

def classifier():
    classifier = Sequential()
    classifier.add(Dense(units = 200, kernel_initializer = 'uniform',
                         activation = 'relu', input_dim = 93))
    classifier.add(Dense(units = 50, kernel_initializer = 'uniform',
                         activation = 'relu'))
    classifier.add(Dense(units = 9, kernel_initializer = 'uniform',
                         activation = 'softmax'))
    classifier.compile(optimizer = 'rmsprop',
                       loss = 'sparse_categorical_crossentropy', metrics = ['accuracy'])
    return classifier
```

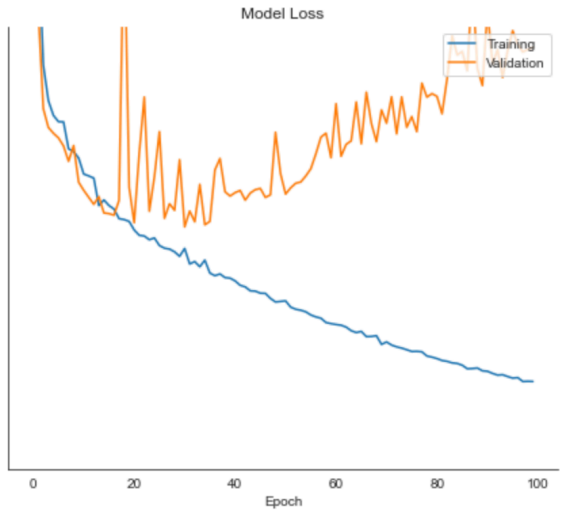
```
classifier = classifier()
history = classifier.fit(X_train, y_train,
                         validation_data=(X_test,y_test),
                         epochs=10, batch_size=1000)
```

```
Epoch 1/10
50/50 [==============================] - 1s 6ms/step - loss: 1.6801 - accuracy: 0.5262 - val_loss: 0.9043 - val_acc
uracy: 0.6745
Epoch 2/10
50/50 [==============================] - 0s 3ms/step - loss: 0.8347 - accuracy: 0.7066 - val_loss: 0.7004 - val_acc
uracy: 0.7418
Epoch 3/10
50/50 [==============================] - 0s 3ms/step - loss: 0.6721 - accuracy: 0.7482 - val_loss: 0.6336 - val_acc
uracy: 0.7548
Epoch 4/10
50/50 [==============================] - 0s 4ms/step - loss: 0.6239 - accuracy: 0.7624 - val_loss: 0.6061 - val_acc
uracy: 0.7667
Epoch 5/10
50/50 [==============================] - 0s 5ms/step - loss: 0.5948 - accuracy: 0.7708 - val_loss: 0.5899 - val_acc
uracy: 0.7669
```

Introduction to Deep Learning

Introduction to Elephas



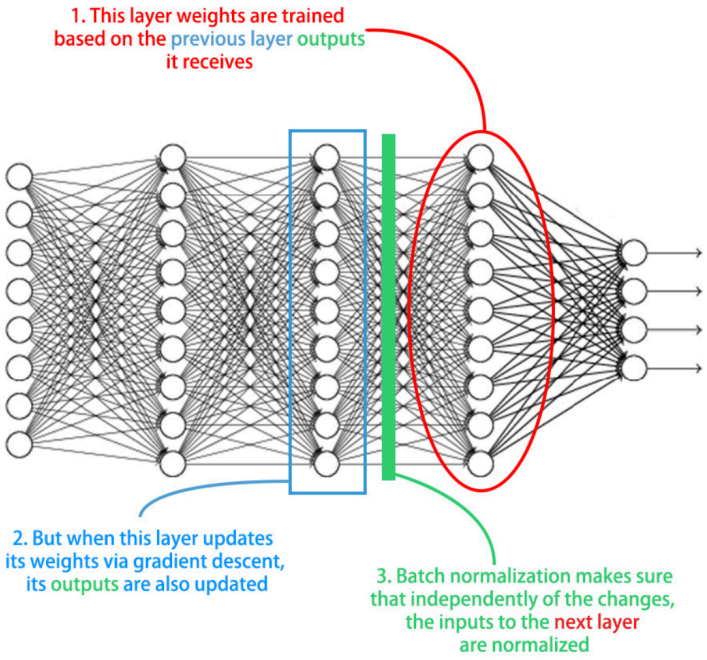


Introduction to Deep Learning

Understanding the Batch Normalization

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. The major advantages are-

- Improved regularization
- Better accuracy
- Faster convergence



Introduction to Deep Learning

Implementing the Batch Normalization

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

def classifier():
    classifier = Sequential()
    classifier.add(Dense(units = 200, input_dim = 93))
    classifier.add(BatchNormalization())
    classifier.add(Dense(units = 50))
    classifier.add(BatchNormalization())
    classifier.add(Dense(units = 9, activation = 'softmax'))
    classifier.compile(optimizer = 'sgd',
                       loss = 'sparse_categorical_crossentropy',
                       metrics = ['accuracy'])
    return classifier

classifier = classifier()
history = classifier.fit(X_train, y_train,
                         validation_data=(X_test,y_test),
                         epochs=10, batch_size=1000)
```

```
Epoch 1/10
50/50 [==============================] - 1s 9ms/step - loss: 2.2095 - accuracy: 0.2796 - val_loss: 1.3212 - val_acc
uracy: 0.6182
Epoch 2/10
50/50 [==============================] - 0s 6ms/step - loss: 1.2439 - accuracy: 0.6377 - val_loss: 1.0857 - val_acc
uracy: 0.6822
Epoch 3/10
50/50 [==============================] - 0s 5ms/step - loss: 1.0551 - accuracy: 0.6848 - val_loss: 0.9826 - val_acc
uracy: 0.7008
Epoch 4/10
50/50 [==============================] - 0s 5ms/step - loss: 0.9625 - accuracy: 0.7005 - val_loss: 0.9226 - val_acc
uracy: 0.7094
Epoch 5/10
50/50 [==============================] - 0s 5ms/step - loss: 0.9072 - accuracy: 0.7147 - val_loss: 0.8828 - val_acc
uracy: 0.7149
Epoch 6/10
50/50 [==============================] - 0s 5ms/step - loss: 0.8636 - accuracy: 0.7203 - val_loss: 0.8541 - val_acc
uracy: 0.7192
```

Introduction to Deep Learning

Applying Callbacks

A callback is a powerful tool to customize the behaviour of a Keras model during training, evaluation, or inference

Some of the major advantages offered by the feature are-

- **Early stopping at minimum loss**: stops training when the minimum of loss has been reached
- **Learning rate scheduling**: dynamically change the learning rate of the optimizer during the course of training

```
from keras.callbacks import EarlyStopping

classifier_es = classifier()
early_stop = EarlyStopping(monitor='val_loss',
                           min_delta=0,
                           patience=4,
                           verbose=0, mode='auto')

history_es = classifier_es.fit(X_train, y_train, validation_data=(X_test,y_test),
                            epochs=100, batch_size=1000, callbacks=[early_stop])
```

Introduction to Deep Learning

Applying Callbacks

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

def classifier():
    classifier = Sequential()
    classifier.add(Dense(units = 200, kernel_initializer = 'uniform', activation = 'relu', input_dim = 93))
    classifier.add(Dense(units = 175, kernel_initializer = 'uniform', activation = 'relu'))
    classifier.add(Dense(units = 150, kernel_initializer = 'uniform', activation = 'relu'))
    classifier.add(Dense(units = 75, kernel_initializer = 'uniform', activation = 'relu'))
    classifier.add(Dense(units = 25, kernel_initializer = 'uniform', activation = 'relu'))
    classifier.add(Dense(units = 9, kernel_initializer = 'uniform', activation = 'softmax'))
    classifier.compile(optimizer = 'rmsprop', loss = 'sparse_categorical_crossentropy', metrics = ['accuracy'])
    return classifier
```

```
from keras.callbacks import EarlyStopping

classifier_es = classifier()
early_stop = EarlyStopping(monitor='val_loss',
                           min_delta=0,
                           patience=4,
                           verbose=0, mode='auto')

history_es = classifier_es.fit(X_train, y_train, validation_data=(X_test,y_test),
                               epochs=100, batch_size=1000, callbacks=[early_stop])
```

Observe that we have specified that the model may train up to 100 epochs. However, by setting the patience= 4, the model will stop training at 50th epoch due the early callback feature.

Introduction to Deep Learning

Cross Validation

Just like the offerings provided by scikit-learn to perform the cross validation while doing model selection, Keras in the same manner provides us with a wrapper that allows to perform cross validation over the performance of our deep learning model.

In this particular example, observe that we are using K-Fold cross validation by means of shuffling our training instances.

```
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import KFold, cross_val_score

# build estimator, making model compatible with sklearn modules
estimator = KerasClassifier(build_fn=classifier, epochs=5, batch_size=1000, verbose=0)

kfold = KFold(n_splits=5, shuffle=True, random_state=1)
%time results = cross_val_score(estimator, X, y, cv=kfold, scoring='accuracy')
print("Accuracy: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

```
CPU times: user 40 s, sys: 13.7 s, total: 53.6 s
Wall time: 13.8 s
Accuracy: 57.79% (14.78%)
```

BTS | Barcelona Technology School

Distributed Deep Learning Model Training

Distributed Model Training

The Big Data Learning Problem

Why existing deep learning frameworks are not efficient?

- In recent years it has been shown that being able to train large deep neural networks on vast amount of data yield state-of-the-art classification performance.
- However, training these models usually takes days, or in some cases, even weeks.
- In order to significantly reduce the training time of these models, Jeff Dean (Google) introduced a new paradigm to train neural networks in a distributed fashion, i.e., model – and data parallelism, which is an initial attempt to tackle this problem.

Distributed Model Training

Model Parallelism

Model Parallelism:

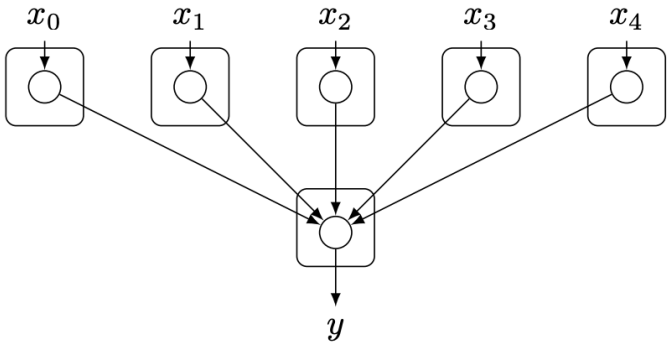
- In model parallelism, a single model is distributed over multiple machines.
- The performance benefits of distributing a deep network across multiple machines mainly depends on the structure of the model.
- Models with a large number of parameters typically benefit from access to more CPU cores and memory, up to the point where communication costs, i.e., propagation of weight updates and synchronization mechanisms, dominate.

Distributed Model Training

Model Parallelism

Model Parallelism:

- In model parallelism, a single model is distributed over multiple machines.
- The performance benefits of distributing a deep network across multiple machines mainly depends on the structure of the model.
- Models with a large number of parameters typically benefit from access to more CPU cores and memory, up to the point where communication costs, i.e., propagation of weight updates and synchronization mechanisms, dominate.



- The aforesaid example discusses a perceptron partitioned using the model parallelism paradigm. In this approach every input node is responsible for accepting the input x_i from some source, and multiplying the input with the associated weight w_i . After the multiplication, the result is sent to the node which is responsible for computing y. Of course, this node requires a synchronization mechanism to ensure that the result is consistent. The synchronization mechanism does this by waiting for the results y depends on.

Distributed Model Training

Data Parallelism

Data Parallelism:

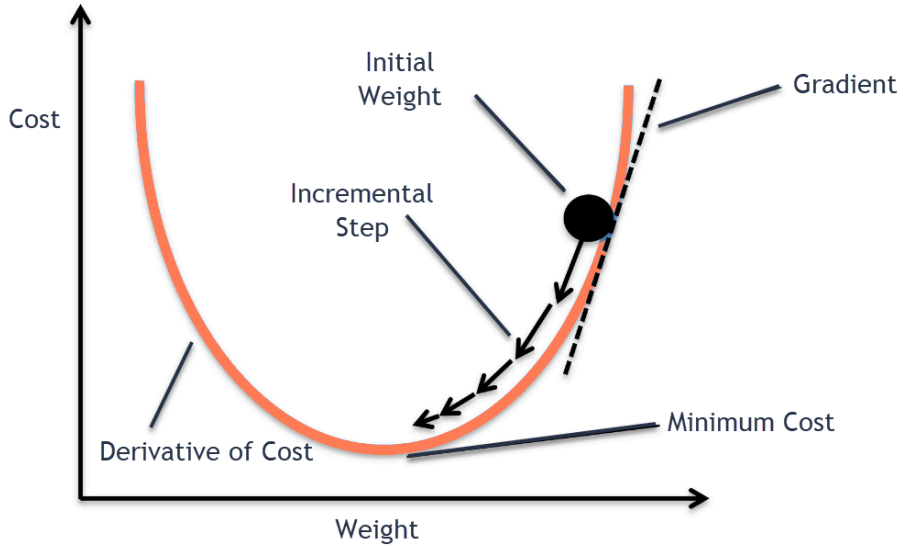
- Data parallelism is an inherently different methodology of optimizing parameters.
- It is a technique to reduce the overall training time of a model.
- In essence, data parallelism achieves this by having n workers optimizing a central model, and at the same time, processing n different shards (partitions) of the dataset in parallel over multiple workers .
- The workers are coordinated in such a way that they optimize the parameterization of a central model.
- The coordination mechanism of the workers can be implemented in many different ways. Nevertheless, a popular approach to coordinate workers in their task to optimize the central objective, is to employ a centralized Parameter Server (PS).
- The sole responsibility of the parameter server is to aggregate model updates coming from the workers (worker commits), and to handle parameter requests (worker pulls).
- In general, there are several approaches towards data parallelism, where some do not require a parameter server. However, all approaches can be categorized into two main groups, i.e., Synchronous Data Parallelism, and Asynchronous Data Parallelism.

Distributed Model Training

Data Parallelism

Gradient Descent:

- Gradient descent is an optimization algorithm often used for finding the weights or coefficients of machine learning algorithms, such as artificial neural networks and logistic regression.
- It works by having the model make predictions on training data and using the error on the predictions to update the model in such a way as to reduce the error.
- The goal of the algorithm is to find model parameters (e.g. coefficients or weights) that minimize the error of the model on the training dataset. It does this by making changes to the model that move it along a gradient or slope of errors down toward a minimum error value. This gives the algorithm its name of “gradient descent.”



Distributed Model Training

Data Parallelism

Stochastic Gradient Descent:

- Stochastic gradient descent, often abbreviated SGD, is a variation of the gradient descent algorithm that calculates the error and updates the model for each example in the training dataset.
- The update of the model for each training example means that stochastic gradient descent is often called an Online Learning Algorithm

Upsides

- The frequent updates immediately give an insight into the performance of the model and the rate of improvement.
- This variant of gradient descent may be the simplest to understand and implement, especially for beginners.
- The increased model update frequency can result in faster learning on some problems.
- The noisy update process can allow the model to avoid local minima (e.g. premature convergence).

Downsides

- Updating the model so frequently is more computationally expensive than other configurations of gradient descent, taking significantly longer to train models on large datasets.
- The frequent updates can result in a noisy gradient signal, which may cause the model parameters and in turn the model error to jump around (have a higher variance over training epochs).
- The noisy learning process down the error gradient can also make it hard for the algorithm to settle on an error minimum for the model.

Distributed Model Training

Data Parallelism

Batch Gradient Descent:

- Batch gradient descent is a variation of the gradient descent algorithm that calculates the error for each example in the training dataset, but only updates the model after all training examples have been evaluated.
- One cycle through the entire training dataset is called a **training epoch**. Therefore, it is often said that batch gradient descent performs model updates at the end of each training epoch.

Upsides

- Fewer updates to the model means this variant of gradient descent is more computationally efficient than stochastic gradient descent.
- The decreased update frequency results in a more stable error gradient and may result in a more stable convergence on some problems.
- The separation of the calculation of prediction errors and the model update lends the algorithm to parallel processing based implementations.

Downsides

- The more stable error gradient may result in premature convergence of the model to a less optimal set of parameters.
- The updates at the end of the training epoch require the additional complexity of accumulating prediction errors across all training examples.
- Commonly, batch gradient descent is implemented in such a way that it requires the entire training dataset in memory and available to the algorithm.
- Model updates, and in turn training speed, may become very slow for large datasets.

Distributed Model Training

Data Parallelism

Mini-Batch Gradient Descent:

- Mini-batch gradient descent is a variation of the gradient descent algorithm that splits the training dataset into small batches that are used to calculate model error and update model coefficients.
- Implementations may choose to sum the gradient over the mini-batch which further reduces the variance of the gradient.
- Mini-batch gradient descent seeks to find a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent. It is the most common implementation of gradient descent used in the field of deep learning.

Upsides

- The model update frequency is higher than batch gradient descent which allows for a more robust convergence,
- avoiding local minima.
- The batched updates provide a computationally more efficient process than stochastic gradient descent.
- The batching allows both the efficiency of not having all training data in memory and algorithm implementations.

Downsides

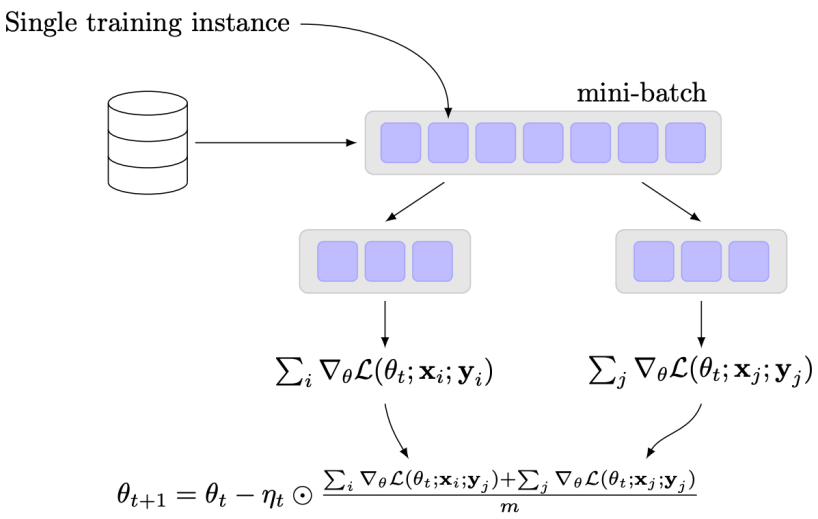
- Mini-batch requires the configuration of an additional “mini-batch size” hyperparameter for the learning algorithm.
- Error information must be accumulated across mini-batches of training examples like batch gradient descent.

Distributed Model Training

Data Parallelism

Data Parallelism:

Synchronous Data Parallelism: Mini-batch parallelism could be viewed as an instance of synchronous data parallelism without a centralized parameter server. Given a mini-batch size of m, we split the mini-batch into several partitions, where a specific worker is responsible for the computation of its own partition. The synchronous nature of this approach lies within the aggregation of the computed gradients, i.e., the results of all workers need to be aggregated, and afterwards averaged in order to integrate the current gradient into the model.

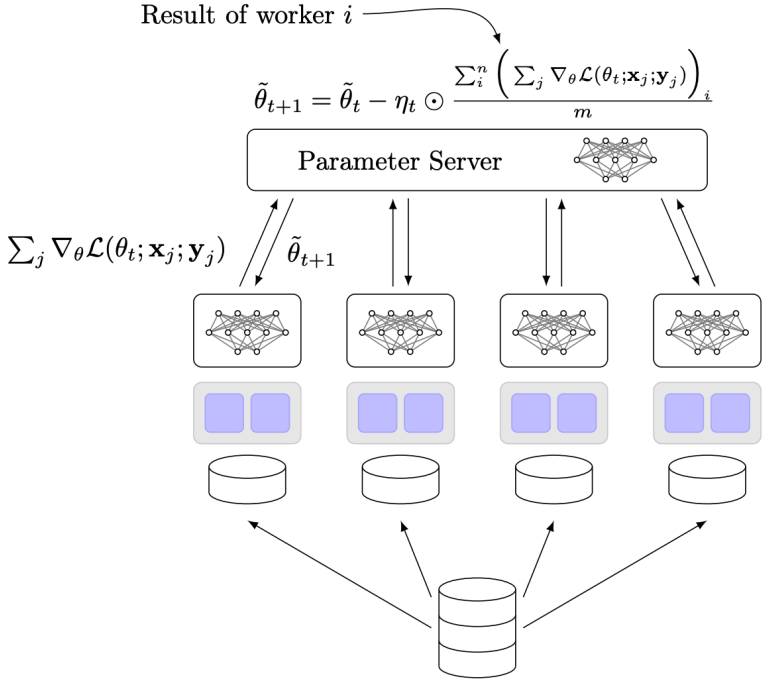


Distributed Model Training

Data Parallelism

Data Parallelism:

Synchronous Data Parallelism: Distributed mini-batch data parallelism with n parallel workers, with a total mini-batch size of m. At the start, the data is split into n partitions, and all workers get a copy of the central model with parameterization. When the training starts, every worker *i* computes the accumulated gradient based on its part of the mini-batch, which is then committed (send) to the parameter server. After the parameter server receives all accumulated gradients, it averages them, and then applies the batched gradient to the model in order to produce. After the new parametrization is available, the workers will be able to pull the fresh copy of parametrization and repeat.



Distributed Model Training

Data Parallelism

Can we eliminate the need for a locking (synchronisation) mechanism to prevent unnecessary waits of the workers?

Asynchronous Data Parallelism: In Asynchronous Data Parallelism workers compute and commit gradients to the parameter server asynchronously. This has as a side-effect that some workers are computing, and thus committing, gradients based on old values. These gradients are called stale gradients.

At the start optimization process, the pull the most recent parameterization from the parameter server. Now all workers start computing the gradients asynchronously based on the pulled parametrization. However, since the parameter server incorporates gradients into the centre variable asynchronously as a simple queuing (FIFO) model, other workers will update the centre variable with gradients based on an older value.

However, surprisingly experiments have shown that removing the synchronization barrier actually allows models to converge*, even when most workers update the central variable using a gradient based on an outdated parameterization of the central variable.

* Read more about the experiments in the recent works described below-

Jeffrey Dean et al. "Large scale distributed deep networks". In: *Advances in neural information processing systems*. 2012, pp. 1223–1231.

Stefan Hadjis et al. "Omnivore: An optimizer for multi-device deep learning on cpus and gpus". In: *arXiv preprint arXiv:1606.04487* (2016).

Sixin Zhang, Anna E Choromanska, and Yann LeCun. "Deep learning with elastic averaging SGD". In: *Advances in Neural Information Processing Systems*. 2015, pp. 685–693.

Distributed Model Training

Introduction to Elephas

- Elephas implements a class of data-parallel algorithms on top of Keras, using Spark's RDDs and data frames.
- Keras Models are initialized on the driver, then serialized and shipped to workers, alongside with data and broadcasted model parameters.
- Spark workers deserialize the model, train their chunk of data and send their gradients back to the driver.
- The "master" model on the driver is updated by an optimizer which takes gradients either synchronously or asynchronously.

Elephas: Distributed Deep Learning with Keras & Spark

Distributed Model Training

Introduction to Elephas

- Elephas implements a class of data-parallel algorithms on top of Keras, using Spark's RDDs and data frames.
- Keras Models are initialized on the driver, then serialized and shipped to workers, alongside with data and broadcasted model parameters.
- Spark workers deserialize the model, train their chunk of data and send their gradients back to the driver.
- The "master" model on the driver is updated by an optimizer which takes gradients either synchronously or asynchronously.

```
from pyspark import SparkContext, SparkConf
conf = SparkConf().setAppName('Elephas_App').setMaster('local[*]')
sc = SparkContext(conf=conf)

from pyspark.sql import SparkSession
spark = SparkSession.builder.master('local[*]')\
.appName('deep-learning').getOrCreate()
```

Distributed Model Training

Introduction to Elephas

- Elephas implements a class of data-parallel algorithms on top of Keras, using Spark's RDDs and data frames.
- Keras Models are initialized on the driver, then serialized and shipped to workers, alongside with data and broadcasted model parameters.
- Spark workers deserialize the model, train their chunk of data and send their gradients back to the driver.
- The "master" model on the driver is updated by an optimizer which takes gradients either synchronously or asynchronously.

```
from pyspark import SparkContext, SparkConf
conf = SparkConf().setAppName('Elephas_App').setMaster('local[*]')
sc = SparkContext(conf=conf)
```

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.master('local[*]')\
.appName('deep-learning').getOrCreate()
```

```
train_df = load_data_frame("../data_path/train.csv")
test_df = load_data_frame("../data_path/test.csv", shuffle=False, train=False)

print("Train data frame:")
train_df.show(10)

print("Test data frame (note the dummy category):")
test_df.show(10)
```

```
Train data frame:
+--------------------+--------+
|            features|category|
+--------------------+--------+
|[0.0,0.0,1.0,0.0,...| Class_3|
|[1.0,0.0,0.0,0.0,...| Class_2|
|[0.0,0.0,0.0,0.0,...| Class_2|
|[0.0,0.0,0.0,0.0,...| Class_8|
|[0.0,0.0,8.0,6.0,...| Class_6|
|[0.0,0.0,0.0,0.0,...| Class_8|
|[0.0,0.0,0.0,0.0,...| Class_2|
|[1.0,0.0,0.0,0.0,...| Class_8|
|[0.0,1.0,6.0,1.0,...| Class_6|
|[0.0,0.0,0.0,0.0,...| Class_2|
+--------------------+--------+
only showing top 10 rows
```

Distributed Model Training

Transforming Labels and Standardising Features

```
from pyspark.ml.feature import StringIndexer

string_indexer = StringIndexer(inputCol="category", outputCol="label")
fitted_indexer = string_indexer.fit(train_df)
indexed_df = fitted_indexer.transform(train_df)
```

```
from pyspark.ml.feature import StandardScaler

scaler = StandardScaler(inputCol="features", outputCol="scaled_features", withStd=True, withMean=True)
fitted_scaler = scaler.fit(indexed_df)
scaled_df = fitted_scaler.transform(indexed_df)
print("The result of indexing and scaling. Each transformation adds new columns to the data frame:")
scaled_df.show(10)
```

```
The result of indexing and scaling. Each transformation adds new columns to the data frame:
+--------------------+--------+-----+--------------------+
|            features|category|label|     scaled_features|
+--------------------+--------+-----+--------------------+
|[0.0,0.0,1.0,0.0,...| Class_3|  3.0|[-0.2535060296260...|
|[1.0,0.0,0.0,0.0,...| Class_2|  0.0|[0.40208999583479...|
|[0.0,0.0,0.0,0.0,...| Class_2|  0.0|[-0.2535060296260...|
|[0.0,0.0,0.0,0.0,...| Class_8|  2.0|[-0.2535060296260...|
|[0.0,0.0,8.0,6.0,...| Class_6|  1.0|[-0.2535060296260...|
|[0.0,0.0,0.0,0.0,...| Class_8|  2.0|[-0.2535060296260...|
|[0.0,0.0,0.0,0.0,...| Class_2|  0.0|[-0.2535060296260...|
|[1.0,0.0,0.0,0.0,...| Class_8|  2.0|[0.40208999583479...|
|[0.0,1.0,6.0,1.0,...| Class_6|  1.0|[-0.2535060296260...|
|[0.0,0.0,0.0,0.0,...| Class_2|  0.0|[-0.2535060296260...|
+--------------------+--------+-----+--------------------+
only showing top 10 rows
```

Distributed Model Training

Introducing a Keras Model

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation
from tensorflow.keras.utils import to_categorical

nb_classes = train_df.select("category").distinct().count()
input_dim = len(train_df.select("features").first()[0])

model = Sequential()
model.add(Dense(512, input_shape=(input_dim,)))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(nb_classes))
model.add(Activation('softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam')
```

Distributed Model Training

Assigning the Keras Model to Elephas Estimator

```
from elephas.ml_model import ElephasEstimator
from tensorflow.keras import optimizers


adam = optimizers.Adam(lr=0.01)
opt_conf = optimizers.serialize(adam)

# Initialize SparkML Estimator and set all relevant properties
estimator = ElephasEstimator()
# The next two paramters come directly from pyspark
estimator.setFeaturesCol("scaled_features")
estimator.setLabelCol("label")
# Provide serialized Keras model to the estimator object
estimator.set_keras_model_config(model.to_yaml())
estimator.set_categorical_labels(True)
estimator.set_nb_classes(nb_classes)
estimator.set_num_workers(1)
estimator.set_epochs(5)
estimator.set_batch_size(128)
estimator.set_verbosity(1)
estimator.set_validation_split(0.15)
estimator.set_optimizer_config(opt_conf)
estimator.set_mode("synchronous")
estimator.set_loss("categorical_crossentropy")
estimator.set_metrics(['acc'])
```

Distributed Model Training

Measuring Model Performance

```
from pyspark.ml import Pipeline

pipeline = Pipeline(stages=[string_indexer, scaler, estimator])
fitted_pipeline = pipeline.fit(train_df)
prediction = fitted_pipeline.transform(train_df)
# prediction = fitted_pipeline.transform(test_df) # <-- The same code evaluates test data.
pnl = prediction.select("label", "prediction")
```

```
from sklearn.metrics import accuracy_score
import numpy as np

metrics_df= prediction.toPandas()
labels= metrics_df["label"].values
predictions= [np.argmax(val) for val in np.array(metrics_df["prediction"].values)]
accuracy_score(labels, predictions)
```

```
0.6420698794401888
```

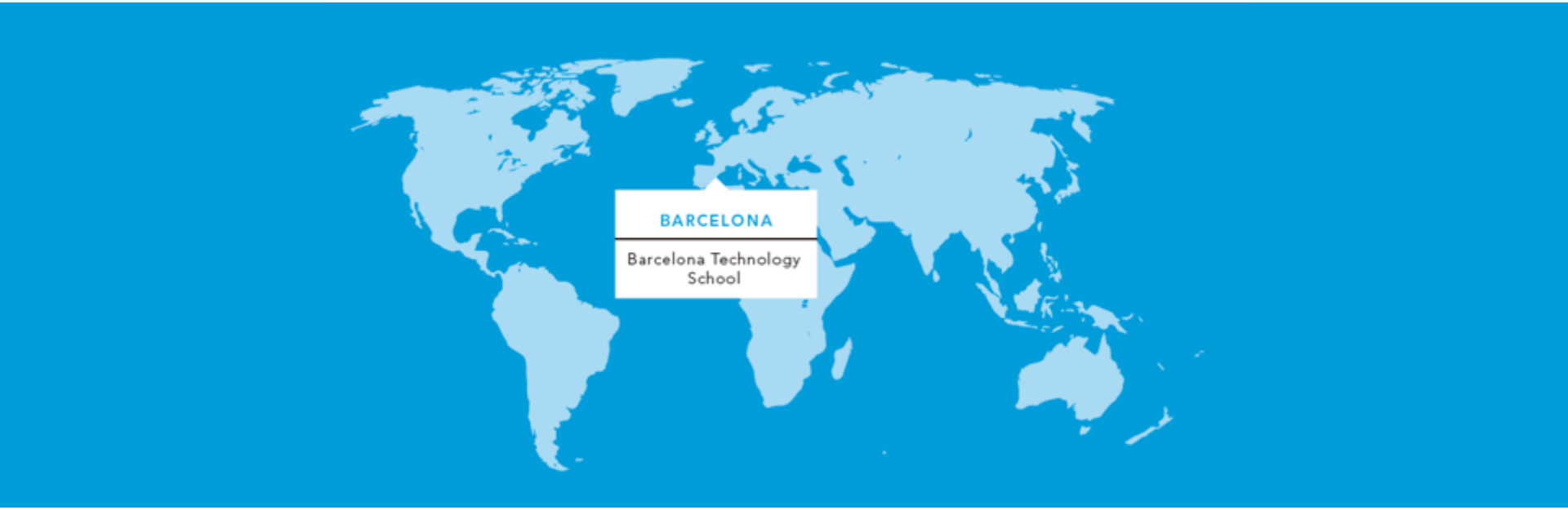
BTS | Barcelona Technology School

References

References

Additional Reading and Academic Content

- CS230: Deep Learning by Stanford University
- CS231N: Convolutional Neural Networks for Visual Recognition
- Deep Learning: Book by Ian Goodfellow, Yoshua Bengio, Aaron Courville
- Neural Networks and Deep Learning: Book by Michael A. Nielson
- Large Scale Distributed Deep Networks: Paper by J. Dean, G.S. Corrado, R. Monga, K. Chen, M. Devin, QV. Le, MZ. Mao, M'A. Ranzato, A. Senior, P. Tucker, K. Yang, and AY. Ng
- HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent: Paper by F. Niu, B. Recht, C. Re, S.J. Wright
- Parallel and Distributed Deep Learning: Paper by Vishakh Hegde and Sheema Usmani from Stanford University
- Some other resource about Distributed Machine and Deep Learning: Various solutions and ideas proposed about the distributed approaches to Machine and Deep Learning
- Elephas: Extension of Keras which allows you to run distributed deep learning models at scale with Apache Spark


BARCELONA
Barcelona Technology School