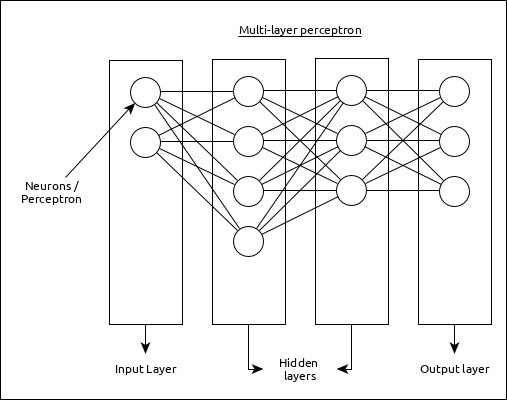
## Multi-Layer Perceptron

Multi-Layer perceptron is the simplest form of ANN. It consists of a single input layer, one or more hidden layer and finally an output layer. A layer consists of a collection of perceptron. Input layer is basically one or more features of the input data. Every hidden layer consists of one or more neurons and process certain aspect of the feature and send the processed information into the next hidden layer. The output layer process receives the data from last hidden layer and finally output the result.



Workflow of ANN

Let us first understand the different phases of deep learning and then, learn how Keras helps in the process of deep learning.

* Collect required data

Deep learning requires lot of input data to successfully learn and predict the result. So, first collect as much data as possible.

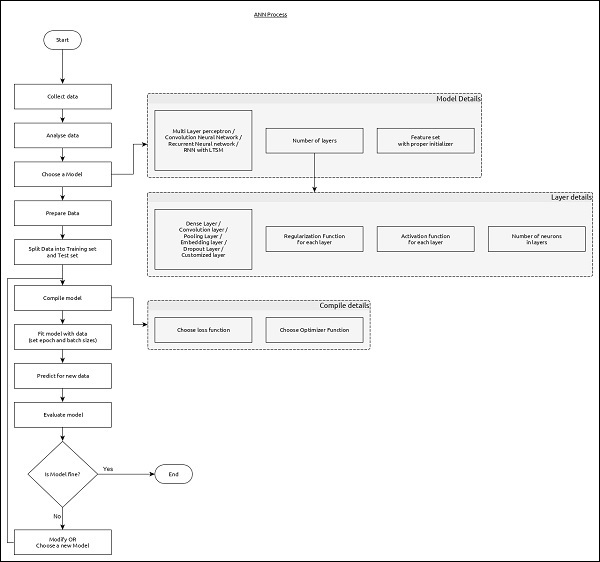
* Analyze data

Analyze the data and acquire a good understanding of the data. The better understanding of the data is required to select the correct ANN algorithm.

* Choose an algorithm (model)

Choose an algorithm, which will best fit for the type of learning process (e.g image classification, text processing, etc.,) and the available input data. Algorithm is represented by **Model** in Keras. Algorithm includes one or more layers. Each layers in ANN can be represented by **Keras Layer** in Keras.

* **Prepare data** − Process, filter and select only the required information from the data.
* **Split data** − Split the data into training and test data set. Test data will be used to evaluate the prediction of the algorithm / Model (once the machine learn) and to cross check the efficiency of the learning process.
* **Compile the model** − Compile the algorithm / model, so that, it can be used further to learn by training and finally do to prediction. This step requires us to choose loss function and Optimizer. loss function and Optimizer are used in learning phase to find the error (deviation from actual output) and do optimization so that the error will be minimized.
* **Fit the model** − The actual learning process will be done in this phase using the training data set.
* **Predict result for unknown value** − Predict the output for the unknown input data (other than existing training and test data)
* **Evaluate model** − Evaluate the model by predicting the output for test data and cross-comparing the prediction with actual result of the test data.
* **Freeze, Modify or choose new algorithm** − Check whether the evaluation of the model is successful. If yes, save the algorithm for future prediction purpose. If not, then modify or choose new algorithm / model and finally, again train, predict and evaluate the model. Repeat the process until the best algorithm (model) is found.



**At the end of this module, you should be able to write code to**:

* Load data for training
* Preprocess the data and convert it to required format
* Build a sequential/functional model
* Compile it using various losses and optimizers as required
* Train the built model
* Evaluate the model
* Optimize the model
* Solve any issues related to keras code

**How we will cover:**

1. High level Intro of Keras and its functionalities
2. Data Loading
3. Preprocessing
4. Model Building
5. Model Evaluation
6. Hyperparameter Tuning
7. Self Help sites or doubts

**Keras Introduction:**

***“Being able to go from idea to result with the least possible delay is key to doing good research****.”*

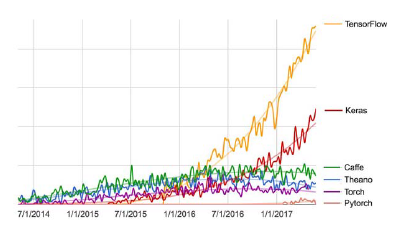
Keras is a high-level neural networks API, capable of running on top of  Tensorflow, Theano, and CNTK. It enables fast experimentation through a high level, user-friendly, modular and extensible API. Keras can also be run on both CPU and GPU.

Keras was developed and is maintained by Francois Chollet and is part of the Tensorflow core, which makes it Tensorflows preferred high-level API

## Keras has the following features :

* Allows for easy and fast prototyping
* Run seamlessly on CPU and GPU
* Supports both convolutional networks(for computer vision) and recurrent networks(for sequence and time-series), as well as the combination of two.
* It supports arbitrary network architectures: multi-input or multi-output models, layer sharing, model sharing and so on. This means Keras is appropriate for building deep learning models, from generative adversarial networks to a neural Turing machine.

Keras is used by around 200,000 users, ranging from academic researchers and engineers at both startups and large companies to graduate students and hobbyist. Keras is used at Google, Netflix, Uber, Microsoft, Square and many startups working on the wide variety of machine learning problems.

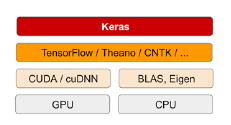


Keras recommend users to switch to *tf.keras*in Tensorflow 2.0, who use multi-backend keras with the tensorflow backend.

**Guiding principles**

* User Friendliness
* Modularity
* Easy Extensibility
* Work with Python

Keras doesn’t handle low-level operations such as tensor manipulations and differentiation. Instead, it relies on a specialized, well-optimized tensor library to do so which serves as the backend engine of Keras. We can use several backend engine for keras, and currently three existing backend implementations are the Tensorflow backend, the Theano backend, and the Microsoft Cognitive Toolkit (CNTK) backend.

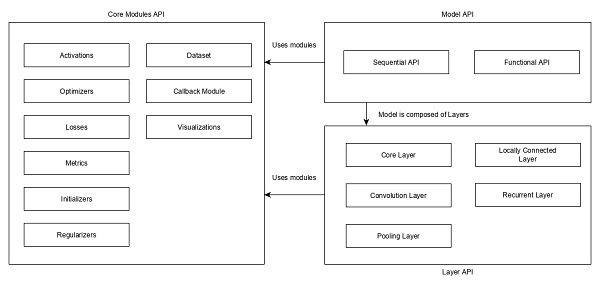


**Keras API can be divided into three main categories** −

* Model
* Layer
* Core Modules

In Keras, every ANN is represented by **Keras Models**. In turn, every Keras Model is composition of **Keras Layers** and represents ANN layers like input, hidden layer, output layers, convolution layer, pooling layer, etc., Keras model and layer access **Keras modules** for activation function, loss function, regularization function, etc. Using Keras model, Keras Layer, and Keras modules, any ANN algorithm (CNN, RNN, etc.,) can be represented in a simple and efficient manner.

The following diagram depicts the relationship between model, layer and core modules −



Let us see the overview of Keras models, Keras layers and Keras modules.

Model

Keras Models are of two types as mentioned below −

**Sequential Model** − Sequential model is basically a linear composition of Keras Layers. Sequential model is easy, minimal as well as has the ability to represent nearly all available neural networks.

**Functional API** − Functional API is basically used to create complex models

Layer

Each Keras layer in the Keras model represent the corresponding layer (input layer, hidden layer and output layer) in the actual proposed neural network model. Keras provides a lot of pre-build layers so that any complex neural network can be easily created. Some of the important Keras layers are specified below,

* Core Layers
* Convolution Layers
* Pooling Layers
* Recurrent Layers

## Core Modules

Keras also provides a lot of built-in neural network related functions to properly create the Keras model and Keras layers. Some of the function are as follows −

* **Activations module** − Activation function is an important concept in ANN and activation modules provides many activation function like softmax, relu, etc.,
* **Loss module** − Loss module provides loss functions like mean\_squared\_error, mean\_absolute\_error, poisson, etc.,
* **Optimizer module** − Optimizer module provides optimizer function like adam, sgd, etc.,
* **Regularizers** − Regularizer module provides functions like L1 regularizer, L2 regularizer, etc.

## ****Data Loading****

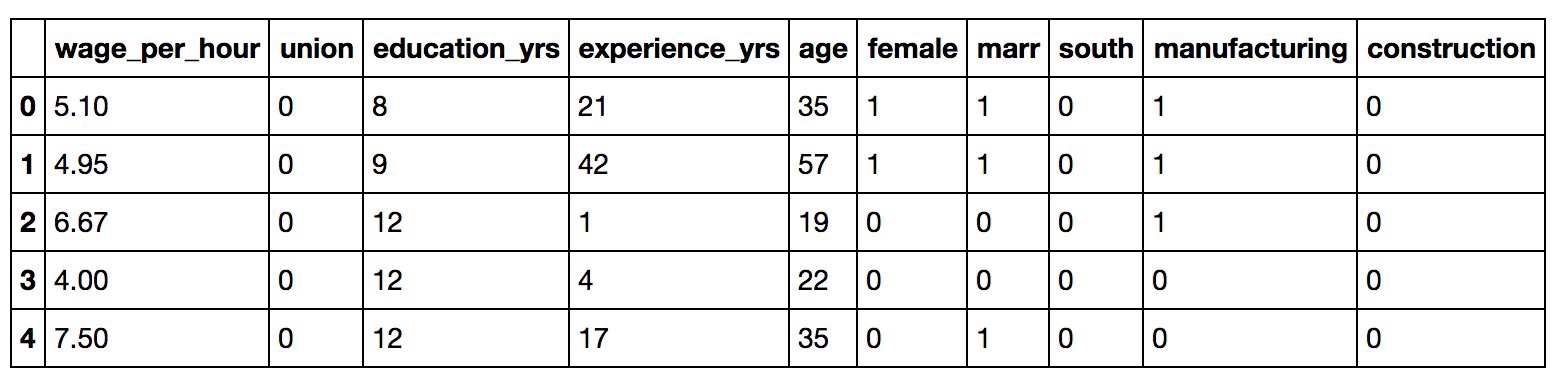
For our regression deep learning model, the first step is to read in the data we will use as input. For this example, we are using the ‘hourly wages’ dataset. To start, we will use Pandas to read in the data. I will not go into detail on Pandas, but it is a library you should become familiar with if you’re looking to dive further into data science and machine learning.

‘df’ stands for dataframe. Pandas reads in the csv file as a dataframe. The ‘head()’ function will show the first 5 rows of the dataframe so you can check that the data has been read in properly and can take an initial look at how the data is structured.

import pandas as pd

#read in data using pandas  
train\_df = pd.read\_csv(‘data/hourly\_wages\_data.csv’)

#check data has been read in properly  
train\_df.head()



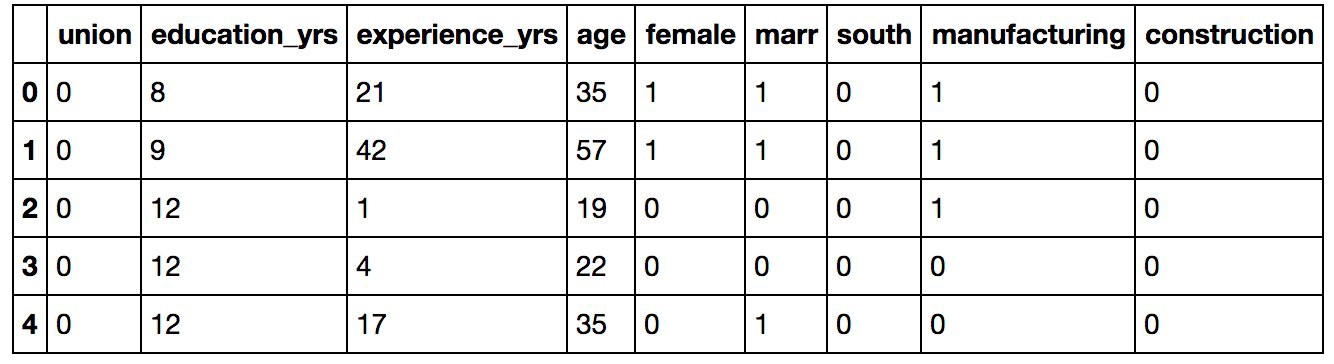
## Preprocessing:

## Split up the dataset into inputs and targets

Next, we need to split up our dataset into inputs (train\_X) and our target (train\_y). Our input will be every column except ‘wage\_per\_hour’ because ‘wage\_per\_hour’ is what we will be attempting to predict. Therefore, ‘wage\_per\_hour’ will be our target.

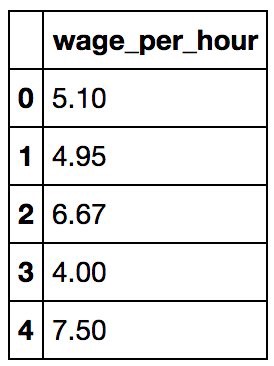
We will use pandas ‘drop’ function to drop the column ‘wage\_per\_hour’ from our dataframe and store it in the variable train\_X’. This will be our input.

#create a dataframe with all training data except the target column  
train\_X = train\_df.drop(columns=['wage\_per\_hour'])  
  
#check that the target variable has been removed  
train\_X.head()



We will insert the column ‘wage\_per\_hour’ into our target variable (train\_y).

*#create a dataframe with only the target column*  
train\_y = train\_df[['wage\_per\_hour']]  
  
*#view dataframe*  
train\_y.head()



## Building the model

Next, we have to build the model. Here is the code:

**from** **keras.models** **import** Sequential  
**from** **keras.layers** **import** Dense#create model  
model = Sequential()  
  
#get number of columns in training data  
n\_cols = train\_X.shape[1]  
  
#add model layers  
model.add(Dense(10, activation='relu', input\_shape=(n\_cols,)))  
model.add(Dense(10, activation='relu'))  
model.add(Dense(1))

The model type that we will be using is Sequential. Sequential is the easiest way to build a model in Keras. It allows you to build a model layer by layer. Each layer has weights that correspond to the layer the follows it.

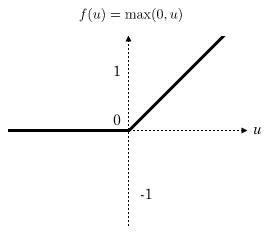
We use the ‘add()’ function to add layers to our model. We will add two layers and an output layer.

‘Dense’ is the layer type. Dense is a standard layer type that works for most cases. In a dense layer, all nodes in the previous layer connect to the nodes in the current layer.

We have 10 nodes in each of our input layers. This number can also be in the hundreds or thousands. Increasing the number of nodes in each layer increases model capacity. I will go into further detail about the effects of increasing model capacity shortly.

‘Activation’ is the activation function for the layer. An activation function allows models to take into account nonlinear relationships. For example, if you are predicting diabetes in patients, going from age 10 to 11 is different than going from age 60–61.

The activation function we will be using is ReLU or Rectified Linear Activation. Although it is two linear pieces, it has been proven to work well in neural networks.



The first layer needs an input shape. The input shape specifies the number of rows and columns in the input. The number of columns in our input is stored in ‘n\_cols’. There is nothing after the comma which indicates that there can be any amount of rows.

The last layer is the output layer. It only has one node, which is for our prediction.

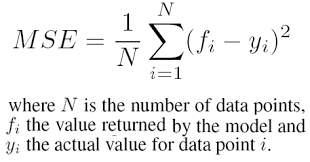
## Compiling the model

Next, we need to compile our model. Compiling the model takes two parameters: optimizer and loss.

The optimizer controls the learning rate. We will be using ‘adam’ as our optmizer. Adam is generally a good optimizer to use for many cases. The adam optimizer adjusts the learning rate throughout training.

The learning rate determines how fast the optimal weights for the model are calculated. A smaller learning rate may lead to more accurate weights (up to a certain point), but the time it takes to compute the weights will be longer.

For our loss function, we will use ‘mean\_squared\_error’. It is calculated by taking the average squared difference between the predicted and actual values. It is a popular loss function for regression problems. The closer to 0 this is, the better the model performed.



#compile model using mse as a measure of model performance  
model.compile(optimizer='adam', loss='mean\_squared\_error')

## Training the model

Now we will train our model. To train, we will use the ‘fit()’ function on our model with the following five parameters: training data (train\_X), target data (train\_y), validation split, the number of epochs and callbacks.

The validation split will randomly split the data into use for training and testing. During training, we will be able to see the validation loss, which give the mean squared error of our model on the validation set. We will set the validation split at 0.2, which means that 20% of the training data we provide in the model will be set aside for testing model performance.

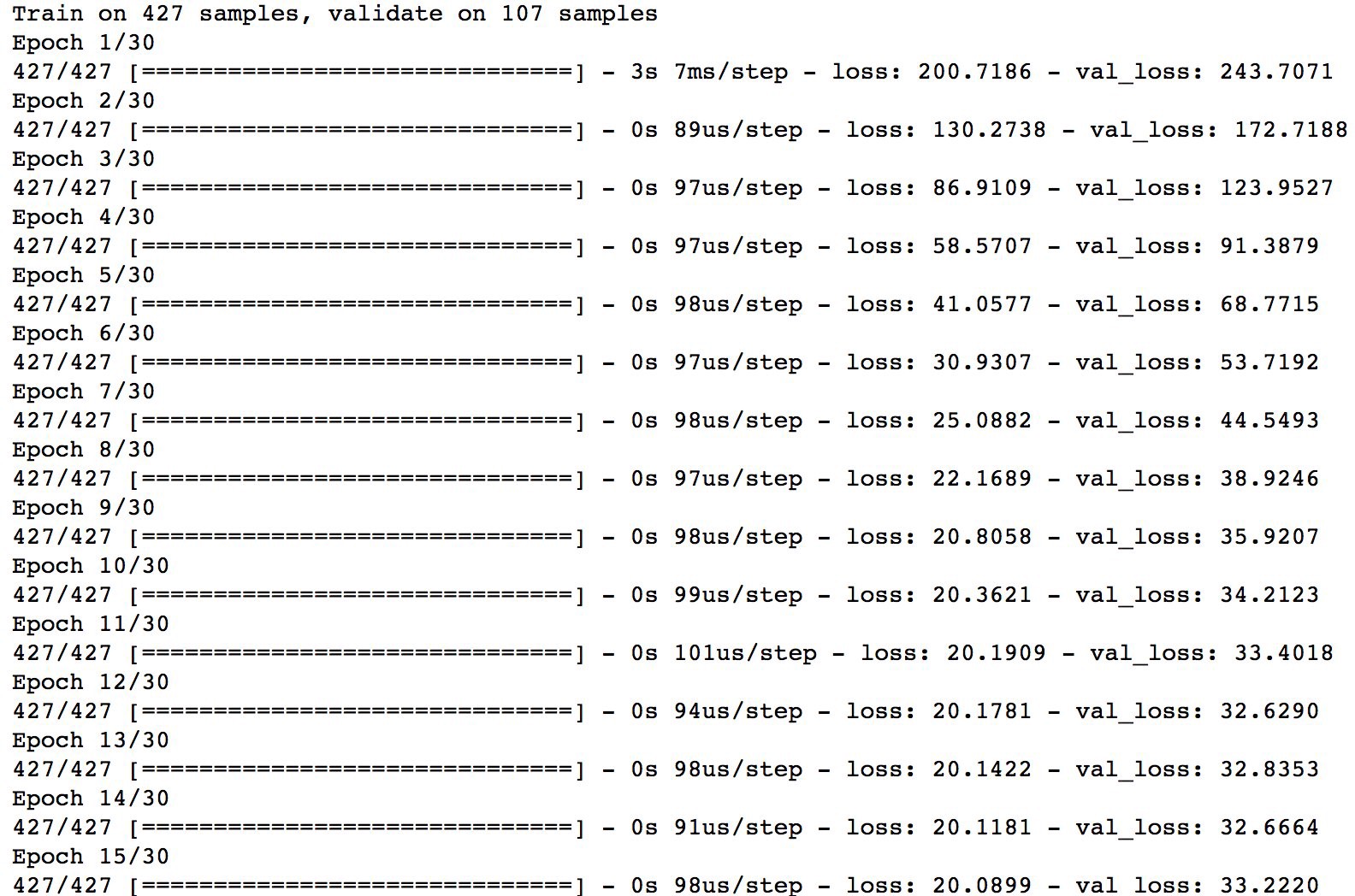
The number of epochs is the number of times the model will cycle through the data. The more epochs we run, the more the model will improve, up to a certain point. After that point, the model will stop improving during each epoch. In addition, the more epochs, the longer the model will take to run. To monitor this, we will use ‘early stopping’.

Early stopping will stop the model from training before the number of epochs is reached if the model stops improving. We will set our early stopping monitor to 3. This means that after 3 epochs in a row in which the model doesn’t improve, training will stop. Sometimes, the validation loss can stop improving then improve in the next epoch, but after 3 epochs in which the validation loss doesn’t improve, it usually won’t improve again.

**from** **keras.callbacks** **import** EarlyStopping

#set early stopping monitor so the model stops training when it won't improve anymore  
early\_stopping\_monitor = EarlyStopping(patience=3)

#train model  
model.fit(train\_X, train\_y, validation\_split=0.2, epochs=30, callbacks=[early\_stopping\_monitor])



## Model capacity

As you increase the number of nodes and layers in a model, the model capacity increases. Increasing model capacity can lead to a more accurate model, up to a certain point, at which the model will stop improving. Generally, the more training data you provide, the larger the model should be. We are only using a tiny amount of data, so our model is pretty small. The larger the model, the more computational capacity it requires, and it will take longer to train.

Let’s create a new model using the same training data as our previous model. This time, we will add a layer and increase the nodes in each layer to 200. We will train the model to see if increasing the model capacity will improve our validation score.

#training a new model on the same data to show the effect of increasing model capacity  
  
#create model  
model\_mc = Sequential()  
  
#add model layers  
model\_mc.add(Dense(200, activation='relu', input\_shape=(n\_cols,)))  
model\_mc.add(Dense(200, activation='relu'))  
model\_mc.add(Dense(200, activation='relu'))  
model\_mc.add(Dense(1))  
  
#compile model using mse as a measure of model performance  
model\_mc.compile(optimizer='adam', loss='mean\_squared\_error')#train model  
model\_mc.fit(train\_X, train\_y, validation\_split=0.2, epochs=30, callbacks=[early\_stopping\_monitor])

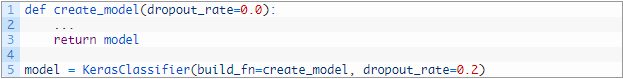
**Hyperparameter tuning using sklearn’s Grid Search CV and sklearn’s wrapper class for Keras**

Keras models can be used in scikit-learn by wrapping them with the **KerasClassifier** or **KerasRegressor** class.

To use these wrappers you must define a function that creates and returns your Keras sequential model, then pass this function to the **build\_fn** argument when constructing the **KerasClassifier** class.

The constructor for the **KerasClassifier** class can also take new arguments that can be passed to your custom **create\_model()** function. These new arguments must also be defined in the signature of your **create\_model()** function with default parameters.

For Ex:

****

## How to Use Grid Search in scikit-learn

Grid search is a model hyperparameter optimization technique.

In scikit-learn this technique is provided in the **GridSearchCV** class.

When constructing this class you must provide a dictionary of hyperparameters to evaluate in the **param\_grid** argument. This is a map of the model parameter name and an array of values to try.

By default, accuracy is the score that is optimized, but other scores can be specified in the **score** argument of the **GridSearchCV** constructor.

By default, the grid search will only use one thread. By setting the **n\_jobs** argument in the **GridSearchCV** constructor to -1, the process will use all cores on your machine. Depending on your Keras backend, this may interfere with the main neural network training process.

The **GridSearchCV** process will then construct and evaluate one model for each combination of parameters. Cross validation is used to evaluate each individual model and the default of 3-fold cross validation is used, although this can be overridden by specifying the **cv** argument to the **GridSearchCV** constructor.

The **best\_score\_** member provides access to the best score observed during the optimization procedure and the **best\_params\_** describes the combination of parameters that achieved the best results.

## How to Tune Batch Size and Number of Epochs

In this first simple example, we look at tuning the batch size and number of epochs used when fitting the network.

The **batch size** in batch gradient descent is the number of patterns shown to the network before the weights are updated. It is also an optimization in the training of the network, defining how many patterns to read at a time and keep in memory.

The **number of epochs** is the number of times that the entire training dataset is shown to the network during training. Some networks are sensitive to the batch size, such as LSTM recurrent neural networks and Convolutional Neural Networks.

Here we will evaluate a suite of different mini batch sizes from 10 to 100 in steps of 20



## How to Tune the Training Optimization Algorithm

Keras offers a suite of different state-of-the-art optimization algorithms.

In this example, we tune the optimization algorithm used to train the network, each with default parameters.

This is an odd example, because often you will choose one approach a priori and instead focus on tuning its parameters on your problem (e.g. see the next example).



## Tips for Hyperparameter Optimization

This section lists some handy tips to consider when tuning hyperparameters of your neural network.

* **k-fold Cross Validation**. You can see that the results from the examples in this post show some variance. A default cross-validation of 3 was used, but perhaps k=5 or k=10 would be more stable. Carefully choose your cross validation configuration to ensure your results are stable.
* **Review the Whole Grid**. Do not just focus on the best result, review the whole grid of results and look for trends to support configuration decisions.
* **Parallelize**. Use all your cores if you can, neural networks are slow to train and we often want to try a lot of different parameters. Consider spinning up a lot of AWS instances.
* **Use a Sample of Your Dataset**. Because networks are slow to train, try training them on a smaller sample of your training dataset, just to get an idea of general directions of parameters rather than optimal configurations.
* **Start with Coarse Grids**. Start with coarse-grained grids and zoom into finer grained grids once you can narrow the scope.
* **Do not Transfer Results**. Results are generally problem specific. Try to avoid favorite configurations on each new problem that you see. It is unlikely that optimal results you discover on one problem will transfer to your next project. Instead look for broader trends like number of layers or relationships between parameters.
* **Reproducibility is a Problem**. Although we set the seed for the random number generator in NumPy, the results are not 100% reproducible. There is more to reproducibility when grid searching wrapped Keras models than is presented in this post.

Assignment:

Apply the hyperparameter tuning on:

1. Learning rate and Momentum
2. Network Weight Initialization
3. Neuron Activation Function
4. Dropout Regularization
5. No. of Neurons in the hidden network