

An introduction to conditional density estimation using the R interface to Keras

Jordan Richards

King Abdullah University of Science and Technology (KAUST)

June 28, 2022



Outline

- ① Background
- ② Installing the R interface to Keras
- ③ Deep learning
 - Basics of neural networks
 - Architecture/types
 - Training
 - Avoiding overfitting
- ④ Building a Keras model
- ⑤ Practicals
 - Logistic regression
 - Single quantile regression
 - GPD regression

What is Keras?

- Keras is a high-level API for fast deep learning developed by Google and written in Python, released in 2015
- Whilst it used to support a number of different back-ends (Theano, MILA; CNTK, Microsoft) it now solely runs on top of Tensorflow
- Tensorflow is a free open-source machine learning (not just DL) library written in Python, C++ and CUDA that does all of the lower-level computations for Keras
- Keras is the most popularly applied deep learning software due to its simple yet powerful framework, followed closely by Facebook's PyTorch. This can also be used in R (see <https://www.rstudio.com/blog/torch/>)

Objectives

The objectives of this short-course are:

- Understand the basics of deep learning and neural networks
- Build and train simple feed-forward prediction and regression models using the R interface to Keras
- Perform conditional density estimation using neural networks

Some suggested reading

- Chollet, F. with Allaire, J. J. (2018). Deep learning with R
- Any of the multitude of Keras for R blogs, see e.g., blogs.rstudio.com, r-bloggers.com, [towardsdatascience](https://towardsdatascience.com), [analaticsvidyha](https://analaticsvidyha.com). Even those that give code written in Python as it's very easy to translate!
- Rodrigeus, F., Pereira, F. C., (2022). Beyond expectation: Deep joint mean and quantile regression for spatiotemporal problems
- For conditional density estimation using deep learning (tailored towards extremes):
 - Cannon, A. J. (2010). A flexible nonlinear modelling framework for nonstationary generalized extreme value analysis in hydroclimatology + Cannon, A. J. (2011). GEVcdn R package
 - Carreau, J., Bengio, Y. (2007). A hybrid Pareto model for asymmetric fat-tailed data: the univariate case
 - Richards, J., Huser, R., (2022). High-dimensional extreme quantile regression using partially-interpretable neural networks: With application to U.S. wildfires

What is conditional density estimation?

Let's say you have a response variable Y with covariates \mathbf{X} :

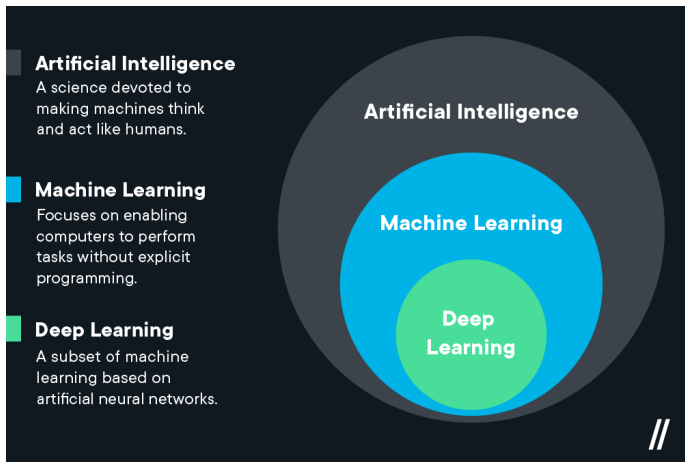
- Conditional density estimation is a generalisation of normal (mean/least-squares) regression - Rather than modelling $\mathbb{E}[Y|\mathbf{X}]$, we model the entire density $f(Y|\mathbf{X})$
- In a parametric framework, we let $Y|\mathbf{X} \sim \mathcal{F}(\boldsymbol{\theta})$ for a parameter set $\boldsymbol{\theta}$ and then let $\boldsymbol{\theta}$ be a function m of observations \mathbf{x} of \mathbf{X}
- Prediction \Leftrightarrow Gaussian density estimation (with fixed σ).
Classification problems \Leftrightarrow Multinomial density estimation
- We want to estimate $m(\mathbf{x})$ using Deep Learning

Installation

First thing's first, let's get Keras installed

- Open *installation.R*
- Download and install Python 3.8.4 from <https://www.python.org/downloads/macos/> (unless you already have a working version of Python ≥ 3.5)
- Install the `keras` and `tensorflow` R packages
- Create a virtual Python environment
- Configure the Rstudio Python interpreter
- Install the latest versions of the Python libraries *tensorflow* and *keras*

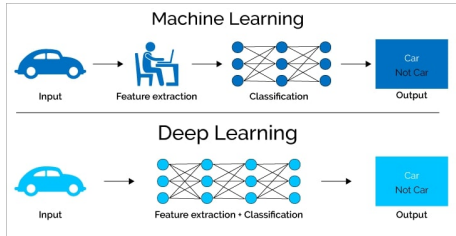
What is deep learning?¹



¹Copyright belongs to
<https://flatironschool.com/blog/deep-learning-vs-machine-learning/>

Deep learning vs. machine learning²

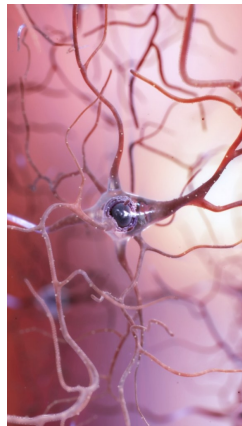
- Use artificial neural networks (ANNs)
- Algorithms are much more complex and require less human intervention \Rightarrow no manual feature extraction required
- Typically requires substantially more data. Gives rise to the concept of transfer learning, i.e., using pre-trained models



²Copyright belongs to

Basics

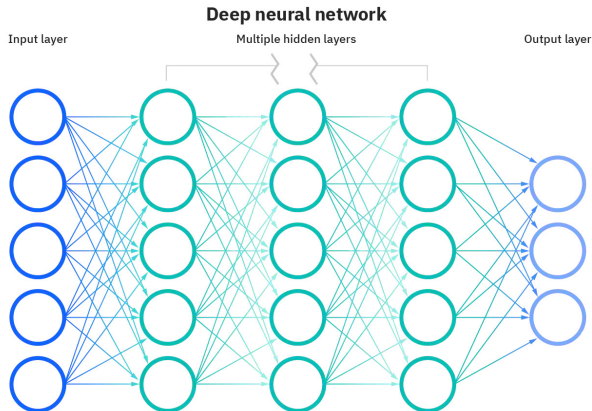
- A neural network is an algorithm designed to mimic the human brain.
- It can be constructed as a directed graph with a single input and a single output, with a dense network of interconnected nodes in-between
- Data is input into the graph and information is extracted at each node (perceptron). An output is provided at the end of the network.
- Similarly brain activity occurs when a stimuli enters the system, information is based through the network via neurons that extra relevant information and this information passes to some area of the brain that forces a response (output)



The nodes in a neural network are arranged in layers:

- A single node may be connected to several nodes in the layer above, from which it receives information, and to the layer below, which it feeds information.
- Most neural networks are “feed forward” - information is passed forward, layer-to-layer, in one direction only
- Different information is extracted at each layer in a hierarchical fashion, i.e., the most important aspects first.
- The “deep” in deep learning refers to the depth of layers in a neural network

Typical structure³



³Copyright belongs to <https://www.ibm.com/cloud/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks>

Typical structure

- A single input layer. Each input would be a single predictor variable in the form of a scalar value, a sequence or an image.
- A single output layer. For prediction or classification problems, we would have a single node here. For CDE, we would have $|\theta|$ nodes.
- A collection of hidden layers. Preferably more than one! In the previous example, each hidden layer has a width of five.
- Calculations are completed at each node in the hidden layer. These are parametrised by weights and biases.
- The calculations within a layer are of a certain "type". We will consider standard, convolutional and recurrent layers.

We refer to the structure of a neural network as its architecture.

Standard “vanilla” MLPs

A standard multi-layered perception uses `layer_dense` in Keras and takes in a vector of values. Let's say that $\mathbf{x} \in \mathbb{R}^d$ is our input - so the previous layer had d outputs! We take some

- Weights: $w_i \in \mathbb{R}$ for $i = 1, \dots, d$
- Bias: $b \in \mathbb{R}$
- Calculate $\sum_{i=1}^d w_i x_i + b$
- And apply an activation function a to get:

$$a \left(\sum_{i=1}^d w_i x_i + b \right).$$










And that's it!

Activation function⁴

The activation function defines the output of the node. There's a range of different functions that you can use (see `keras_build.rmd`), but typically they must satisfy two properties:

- Nonlinear! If all nodes had linear activation, this would just be a linear regression model. A two-layered NN with non-linear activations is a universal function approximator.
- Continuously differentiable. Needed for gradient-based optimisation.

You can think of the activation function as an analogue to the link function in regression. In fact, a single-layered NN is simply a generalised linear regression model. We can think of CDE using NNs as a hierarchical generalised linear model.

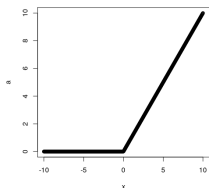
⁴https://en.wikipedia.org/wiki/Activation_function         

ReLU activation

An activation function that's popularly applied is the rectified linear unit:

- Older neural networks relied on sigmoid or tanh activation functions
- These suffered from the *vanishing gradient problem*⁵, which made it difficult to effectively train deep-layered networks
- The Relu replaced these. It has the form

$$a(x) = \max\{x, 0\}$$

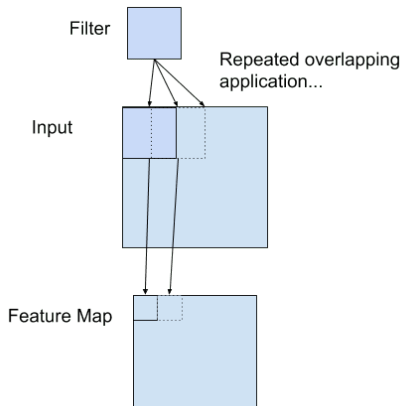


- The neuron “activates” only if enough information passes through (think pain receptors)
- One drawback: not differentiable at zero

⁵The networks we will be considering will not be overly deep, so this problem does not occur. For details, see <https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/>

Convolution layers⁶

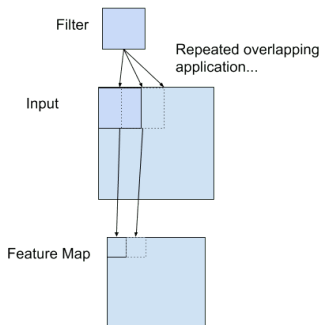
A convolution layer acts on an image, not a vector of inputs. The image can be 1D (e.g., text), 2D or 3D (e.g., a video or 3D model)



- A convolution filter/kernel of a certain dimension is passed over the image
- The filter starts in a corner of the image. A convolution is performed, and then the filter moves one "stride". Once it reaches the edge, it moves one stride down.

⁶Copyright belongs to <https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/>

Convolution layers⁷



- After the filter has passed over the entire image, a feature map will be produced. The dimension of the feature map will be determined by the filter dimension and the stride
- An image can have multiple "channels", i.e., different colours, inputs, each with its own filter
- The feature maps for the different channels are added, a bias is introduced and then a is applied
- A layer can have multiple filters, similar to nodes in a dense network

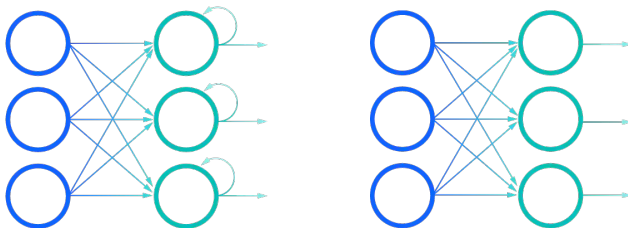
⁷Copyright belongs to <https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/>

Convolutions

- Each filter has an array of weights corresponding to its dimension. A 3×4 filter will have a 3×4 matrix of weights
- A convolution is simply the dot product of the filter weights and the image values
- Unless the filter has all dimensions one and a stride of one, the outputted feature map will have a smaller dimension than the input images. This can be rectified using padding, i.e., padding the dimensions of the inputs with zeroes.
- Very good for capturing spatial characteristics in data

Recurrent layers⁸

A recurrent layer takes in as input a sequence of vectors. Let's say that the input is $\mathbf{x}_t \in \mathbb{R}^d$ for $t = 1, \dots, T$.



Once an input enters the layer, it becomes stuck in a recurrence loop - the output from the computation using the first input vector \mathbf{x}_1 becomes an input for the computation on \mathbf{x}_2 .

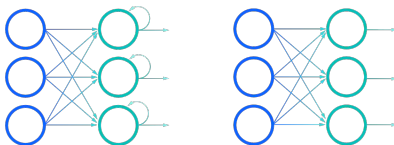
⁸Copyright belongs to

Recurrent layers⁹

Let's consider a very basic example of a recurrent layer. The output from the first input \mathbf{x}_1 is $a(m_1 + b)$ where

$$m_1 = \sum_{i=1}^d x_{i,1} w_i.$$

The output $a(m_1 + b)$ moves to the next layer, but the information m_1 returns back into the same layer.



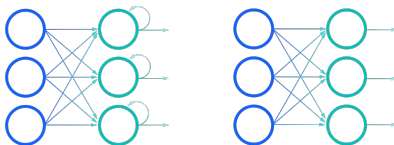
⁹Copyright belongs to

Recurrent layers¹⁰

The output from the second input \mathbf{x}_2 is $a(m_2 + b)$ where

$$m_2 = \sum_{i=1}^d x_{2,1} w_i + m_1 + b.$$

So now the output $a(m_2 + b)$ contains information from the previous vector in the sequence! And this procedure is repeated for the entire sequence and for all nodes.



¹⁰Copyright belongs to

Some points...

- These simple layers can be implemented using `layer_simple_rnn`
- Generally not used anymore due to numerical stability problems, see vanishing/exploding gradients. People now use Long Short-Term Memory layers¹¹. The intuition is the same, but these layers have "forget gates" which allow them to forget useless information.
- I illustrated a "vanilla" recurrent layer. We can have recurrent convolutional layers instead (see e.g., `layer_conv_lstm_2d`), which captures temporal structure within sequences of images
- Unlike the previous layers, computations for recurrent layers cannot be done in parallel. This makes training more computationally demanding and not suitable for laptop-based work. I've not included an explanation of their implementation in Keras, but I can provide help/code on request.

¹¹S. Hochreiter; J. Schmidhuber (1997). "Long short-term memory". *Neural Computation*. 9 (8): 1735–1780.

Loss

Training a neural network is done by optimizing some loss function. Let's say we have N observations of the response y_1, y_2, \dots and the neural network predicts some output \hat{y}_1, \hat{y}_2 (could be a vector!). We write the loss function as $l(y, \hat{y}), \dots$. Whilst the NN architecture can be very standard for any problem, the loss function must be very specific!

Examples include:

- MSE: $(1/N) \sum_{i=1}^N (y_i - \hat{y}_i)^2$ (mean prediction)
- MAE: $(1/N) \sum_{i=1}^N |y_i - \hat{y}_i|$ (median prediction)
- Binary cross-entropy (if $\hat{y}_i \in (0, 1)$ and $y_i = 0$ or 1):
 $-(1/N) \sum_{i=1}^N (y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i))$. Classification, can be extended to more than two classes.

For CDE, we will use the negative log-likelihood as the loss function. Here \hat{y} will be the parameters for distribution we are fitting.

Training

The loss function is minimised using a form of gradient descent. Neural networks are trained for a finite number of epochs (iterations). Let's say all of the trainable weights and parameters for our NN are contained in some set Θ . For epoch i :

- ① We go forward through the network to compute \hat{y} given the current state of the network. This allows us to evaluate $l(y, \hat{y})$.
- ② We move back through the network to compute (exactly) $\nabla l(y, \hat{y})$ w.r.t Θ .
- ③ We update the parameters using

$$\Theta^{(i+1)} = \Theta^{(i)} - \lambda^{(i)} \nabla l(y, \hat{y}),$$

with the product taken componentwise and where $\lambda^{(i)}$ are a set of learning rates.

Training (cont.)

- The learning rates are quite important. Large rates will cause the optimization to diverge; small rates may cause converge to local minima. Algorithms exist for adaptive tuning of λ (hence the superscript (i)), most notably RMSprop¹² and ADAM¹³
- Training using gradient descent can be quite inefficient. Instead we would use Mini-batch gradient descent. First split the data into batches. Within an epoch, parameters are updated for each batch; the partial derivatives of the loss are computed exactly for the batch (which gives an approximation to the true pd's)
- Training using MbGD is more efficient, but the loss is not guaranteed to decrease for each epoch (we will see this later!)
- The extreme case with a batch size of 1 is Stochastic Gradient Descent (although the names are used interchangeably)

¹²Unpublished Coursera course by Geoff Hinton

https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

¹³<https://arxiv.org/abs/1412.6980> with over 111000 citations!

Backpropagation

Often when optimising particularly difficult functions, we might need to approximate the partial derivatives. For neural networks, we compute $\nabla l(y, \hat{y})$ through backpropagation. This involves moving backwards through the network and using the chain rule to calculate the derivatives at each layer. As we have constructed the neural network output as a function of differentiable functions, this is very easy to do!

Illustrating back propagation

Let's take a simple illustration. Say we have a network with two hidden layers with widths four, all of the biases are equal to zero and a single output. Then we can construct

$$\begin{aligned}\hat{y} &= a_3 \left(\sum_{i=1}^4 w_{3,i} m_{2,i} \right) = a_3 \left(\sum_{i=1}^4 w_{3,i} a_2 \left\{ \sum_{i=1}^4 w_{2,i} m_{1,i} \right\} \right) \\ &= a_3 \left(\sum_{i=1}^4 w_{3,i} a_2 \left\{ \sum_{i=1}^4 w_{2,i} a_1 \left[\sum_{i=1}^d w_{1,i} x_i \right] \right\} \right)\end{aligned}$$

The goal is to calculate all $\partial l(y, \hat{y}) / \partial w_{3,i}$, $\partial l(y, \hat{y}) / \partial w_{2,i}$ and $\partial l(y, \hat{y}) / \partial w_{1,i}$.

Illustrating back propagation

Start at the beginning. We know that

$$dI(y, \hat{y})/d\hat{y}$$

will be simple to compute (especially the standards) if we have constructed $I(y, \hat{y})$ from differentiable functions. Similarly we should also know

$$a'_3(x) = da_3(x)/dx, \quad a'_2(x) = da_2(x)/dx, \quad \text{and} \quad a'_1(x) = da_1(x)/dx.$$

Now move back through the network. From the final layer, we have

$$\frac{\partial I(y, \hat{y})}{\partial w_{3,i}} = \frac{\partial I(y, \hat{y})}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_{3,i}} = \frac{\partial I(y, \hat{y})}{\partial \hat{y}} m_{2,i} a'_3 \left(\sum_{i=1}^4 w_{3,i} m_{2,i} \right).$$

Illustrating back propagation

From the second layer, we have

$$\begin{aligned}\frac{\partial l(y, \hat{y})}{\partial w_{2,i}} &= \frac{\partial l(y, \hat{y})}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial m_{2,i}} \frac{\partial m_{2,i}}{\partial w_{2,i}} = \frac{\partial l(y, \hat{y})}{\partial \hat{y}} w_{3,i} a_3' \left(\sum_{i=1}^4 w_{3,i} m_{2,i} \right) \frac{\partial m_{2,i}}{\partial w_{2,i}} \\ &= \frac{\partial l(y, \hat{y})}{\partial \hat{y}} w_{3,i} a_3' \left(\sum_{i=1}^4 w_{3,i} m_{2,i} \right) \frac{\partial m_{2,i}}{\partial w_{2,i}} m_{1,i} a_2' \left(\sum_{i=1}^4 w_{2,i} m_{1,i} \right),\end{aligned}$$

and so on and so forth. As long as we choose differentiable loss and activation functions, backpropagation can be performed very efficiently! When writing custom functions for Keras, we must use the `tensorflow` or Keras backend functions; these have known derivatives that `tensorflow` can easily call.

Training/validation/testing

Due to their large number of parameters, neural networks are prone to overfitting. It's necessary to use validation/testing to avoid this.

- First, we subset the data into training, validation and testing. Opinions differ, but the standard is 80 – 10 – 10
- Each set has a different purpose:
 - The model is **trained** on the training data. This is used to optimise the weights, biases and other hyper-parameters
 - We then **validate** the model by getting predictions for the validation set and evaluating validation loss. This will give us some insight as to whether or not overfitting is occurring. Typically the validation loss is used for model selection
 - To truly get an unbiased evaluation of the model fit and to compare amongst different models, we should **test** the model on previously unseen data
- Some literature uses test and validation interchangeably, e.g., in cross-validation, the validation (hold-out) set is actually a test set because the model never sees it for training

That's the end of the first session.
Thanks for listening!

Overview

We will now put your new practical skills to the test!

- I have simulated 10000 observations of a response Y on a 10×12 grid. The distribution of the response is dependent on a predictor set $\mathbf{X} \in \mathbb{R}^{10}$. Observations are replicates from a GP on the same 10×12 grid. The unknown function that connects the predictors to the parameters of the response distribution is highly non-linear.
- You will build and train a neural network to estimate the conditional density $f(Y|\mathbf{X})$. Choose your own architecture, optimisation scheme, validation sets and try to build the best predictive model possible
- 1000 observations of Y and \mathbf{X} have been left out for testing. At the end we will test everyone's models using this hold-out set, and see who has the best model!

Some notes:

- There's no temporal structure in the data, so using an RNN is unnecessary. This also means a simple validation scheme is all that's necessary
- There is spatial structure in the predictor set, so convolution layers will perform well here. However, you will have to consider the extra time/difficulty in training due to the increased number of parameters, as well as the increased risk of overfitting (and hence poor extrapolation)
- You will have to write your own custom loss function to perform both quantile and GPD regression

Logistic regression

- Load `logistic_train_df.Rdata`
- Build a model to estimate $\Pr\{Y = 0\}$ for the response using the binary cross-entropy loss
- At the end, we will load the test data in `logistic_test_df.Rdata`
- Use your model to predict the probabilities for the test data and then evaluate the binary cross-entropy loss given those predictions

Non-parametric quantile regression

- Load `qregress_train_df.Rdata`
- Build a model to estimate the 90% quantile for the response using the tilted loss function. For the τ -quantile, this is

$$l(y, \hat{y}) = (1/N) \sum_{i=1}^N \max\{\tau(y_i - \hat{y}_i), (\tau - 1)(y_i - \hat{y}_i)\}.$$

- At the end, we will load the test data in `qregress_test_df.Rdata`
- Use your model to predict the quantile for the test data and then evaluate the tilted loss given those predictions
- Save the train/test quantile estimates as `pred_u_train/pred_u_test` (We will use these for fitting a GPD later!)

GPD regression

- For the data from `qregress_train_df.Rdata`, build a model to fit a GPD above your estimated 90% quantile
- The predicted scale and shape parameters will be used to evaluate the TWCRPS for the test data. For a sequence of increasing thresholds $\{v_1, \dots, v_{n_v}\}$ and weight function $w(x)$, the TWCRPS is

$$\sum_{i=1}^N \sum_{j=1}^{n_v} w(v_j) [\mathbb{I}\{y_i \leq v_j\} - \hat{p}_i(v_j)]^2,$$

where $\hat{p}_i(v_j)$ denotes the predicted probability $\Pr\{Y_i < v_j\}$. The weight function is $w(x) = 1 - (1 + (x + 1)^2)^{-1/6}$ and puts more weight on extreme values.

Thanks for your attention!