Machine Learning in Finance
Section I3
Lecture 10

Michael G Sotiropoulos

NYU Tandon
Deutsche Bank

12-Apr-2019

Contents

In the function approximation approach to machine learning, an ML algorithm is a method for constructing a mapping from inputs to outputs, $x \to \hat{y}$.

An **A**rtificial **N**eural **N**etwork (ANN) is a method of building ML algorithms. An ANN computes the mapping $x \to \hat{y}$ by breaking it down into smaller operations. Operations are interconnected so that the outputs of one operation become inputs to several others. ANNs are therefore **connectionist systems**.

- **Artificial**: to be distinguished from biological neural networks, from which ANNs have been initially inspired, and with which they are compared.
- **Neural**: the interconnected operations (nodes or units) are called neurons. Connection between neurons correspond to the synapses of the brain.
- **Network**: the connections among nodes define a network topology.

The network topology of an ANN typically looks like layers of nodes, with connections between adjacent layers (like a layered cake with filling).

- The first layer is connected to the inputs and the last to the outputs.
- The in-between layers are called **hidden layers**.

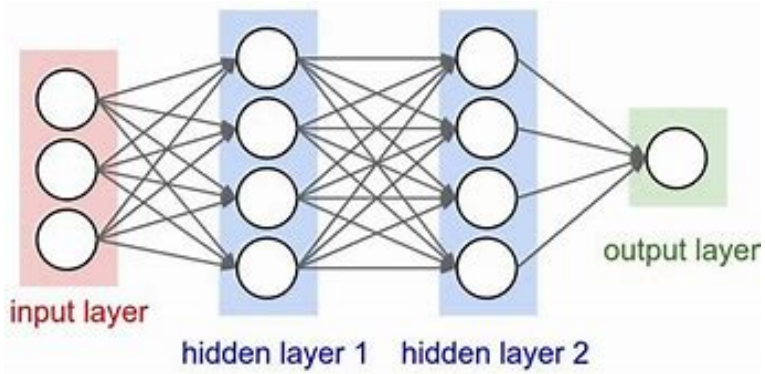

Figure: A feed-forward dense network with two hidden layers.

Mathematically, an ANN is a concrete implementation of the universal approximation theorem (Cybenko 1989).

- A feed-forward network with a single hidden layer of K nodes can approximate *any* continuous function over a compact subset of $\mathbb{R}^J$.

Consider a continuous function $f(x)$ that maps vectors of size J to scalars ($\mathbb{R}^J \rightarrow \mathbb{R}$), defined over a compact set (say a J-hypercube).

- This defines a model for predicting one output y from J input attributes x.

According to the theorem, the function can be approximated as

$$\hat{y} = \hat{f}(x) = \sum_{k=1}^{K} \phi \left(\sum_{j=1}^{J} x_j w_{j,k} + b_k \right) v_k + c \tag{1}$$

$j = 1 \dots J$	is the input vector component index
$k = 1 \dots K$	is the hidden node index
$w_{j,k}$, b_k	are the weight matrix and bias vector of the hidden layer
ϕ	is a nonlinear activation function, applied component-wise
v_k, c	are the weights and bias of the output layer
$\left\lvert f(x) - \hat{f}(x) \right\rvert^p$	is the loss function, i.e the size of the residuals

- A **deep** neural net is an ANN with more than one hidden layer, i.e. it does several iterations of the sequence: transform activate.
- ANN supervised learning is the process of fitting the weights w, v and biases b, c given inputs x and true outputs y.
- This is done via an optimization algorithm that minimizes the loss.

One of the earliest ANNs is the Perceptron (Rosenblatt 1957).
In its simplest form it consists of one input layer and one output layer.

- Each input x_j is associated with a weight w_j.
- There is also a bias c. This can be the represented as the weight of a constant input node that always returns 1.
- The output layer computes the linear combination of the input $f_\Sigma = \boldsymbol{x}^T \cdot \boldsymbol{w} + c$
- The activation function $\sqcap$ is the Heaviside step function $\Theta(z) = I(z \geq 0)$ Final output: $y = \Theta\left(\boldsymbol{x}^T \cdot \boldsymbol{w} + c\right)$

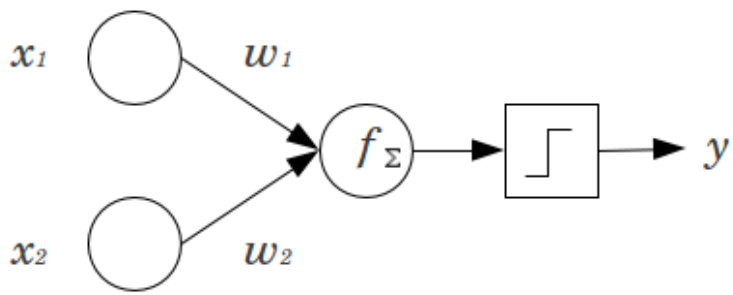

Figure: Perceptron with two input attributes and one output

This is a linear binary classifier. If the linear combination of the input $\boldsymbol{x}^T \cdot \boldsymbol{w}$ exceeds a threshold $-c$, it predicts the positive class (1) , else the negative (0).

The learning task is to find $\boldsymbol{w}$ such as misclassification error is minimized.
Rosenblatt's training algorithm is similar to SGD with η the learning rate.
The training algorithm is iterative (one observation at a time).

- For binary classification it is

$$w_j^{\text{next}} = w_j^{\text{curr}} + \eta\,(\hat{y} - y)\,x_j \tag{2}$$

- For K binary classes (J inputs connected to K outputs) it is

$$w_{j,k}^{\text{next}} = w_{j,k}^{\text{curr}} + \eta\,(\hat{y}_k - y_k)\,x_j \tag{3}$$

If the training set is linearly separable, the algorithm converges to a solution.

- The perceptron was (unfairly?) criticized as being incapable of learning non-linearly separable sets, like the XOR (exclusive or) function.
- This criticism is also true for logistic regression and SVMs. In fact, SVM's were developed to improve the perceptron, because they find a unique maximum margin decision boundaries.

Enthusiasm waned, until the mid-80's, when multi-layer perceptrons and the backpropagation training algorithm were introduced.
Since then, technical progress in hardware and commercial interest in consumer electronics and services have led to today's ANN hyperactivity.

ANN Anatomy: Construction (I)

Construction and training of an ANN is done in the following steps:

1. **Definition of the network topology**
   - Select the number, type, shape and initialization of the hidden layers to use, and arrange them sequentially between input and output
   - Select the activation functions to be used between layers

After this phase is completed we have defined a sequential, dense ANN

2. **Compilation or model creation**
   - Select the type of optimizer to use
   - Select the loss function to use for supervised learning
   - Select any extra metrics that need to be collected during training

After this phase is completed we have created an ANN model that can learn

3. **Training or fitting**
   - Reshape/preprocess inputs/outputs and bind them to the model
   - Decide how many passes to do over the data set (**epochs**) during fitting
   - Decide the size of **mini-batches** per epoch

Call the fit function and monitor its progress.

Model evaluation and tuning is similar to all other ML algos

- Evaluate the model out of sample
- Tune the model by changing topology, activations, optimizer parameters

The main difference between the ANN and the classic ML workflow is that ANN models are built in two steps (definition/compilation) instead of one step. This is because there is too much variability in the way ANNs can be built, which would not easily fit in one constructor function call.

The main benefit of the two step process, is that the definition phase creates a **computational graph**.

- A computational graph defines a mapping from inputs to outputs, using placeholders (symbolic variables) for data and nodes for operations.

The graph is created at runtime (unlike standard functions defined in code)
To evaluate the graph, we bind inputs to the variables and call **eval**

Most importantly, a computational graph allows us to generate the derivatives of the function defined by the graph, at runtime using autodiff methods.

Fitting ANN parameters means running optimizers.
Almost all optimizers require computation of gradients (first order derivatives).
Some optimizers may require Hessian matrices (second order derivatives).

Automatic differentiation can be done using the following methods

1. **Numerical**: $f'(x) = \lim_{\epsilon \to 0} \left(f\left(x + \epsilon\right) - f\left(x\right) \right) / \epsilon$
   Requires two function evaluations and division by a small number ϵ.
   Simple to implement, works with all functions, but low precision
2. **Symbolic**: rules engine that computes derivatives like a human
   $\partial x / \partial x = 1$, $\partial x / \partial y = 0$, $\partial x^n / \partial x = n x^{n-1}$, product rule, chain rule, ...
   Exact, but may create large graphs, does not work with every function
3. **Forward autodiff**: First order Taylor expansion using dual numbers.
   A dual is represented as a pair of an "ordinary" and an "infinitesimal".
   $(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\,\epsilon$
   $(a + b\epsilon) \times (c + d\epsilon) = ac + (bc + ad)\,\epsilon$
   A forward pass through the computational graph evaluates
   $f\left(x + \epsilon\right) = f\left(x\right) + f'\left(x\right)\epsilon$
4. **Reverse autodiff**: repeated application of chain rule, see next figure.

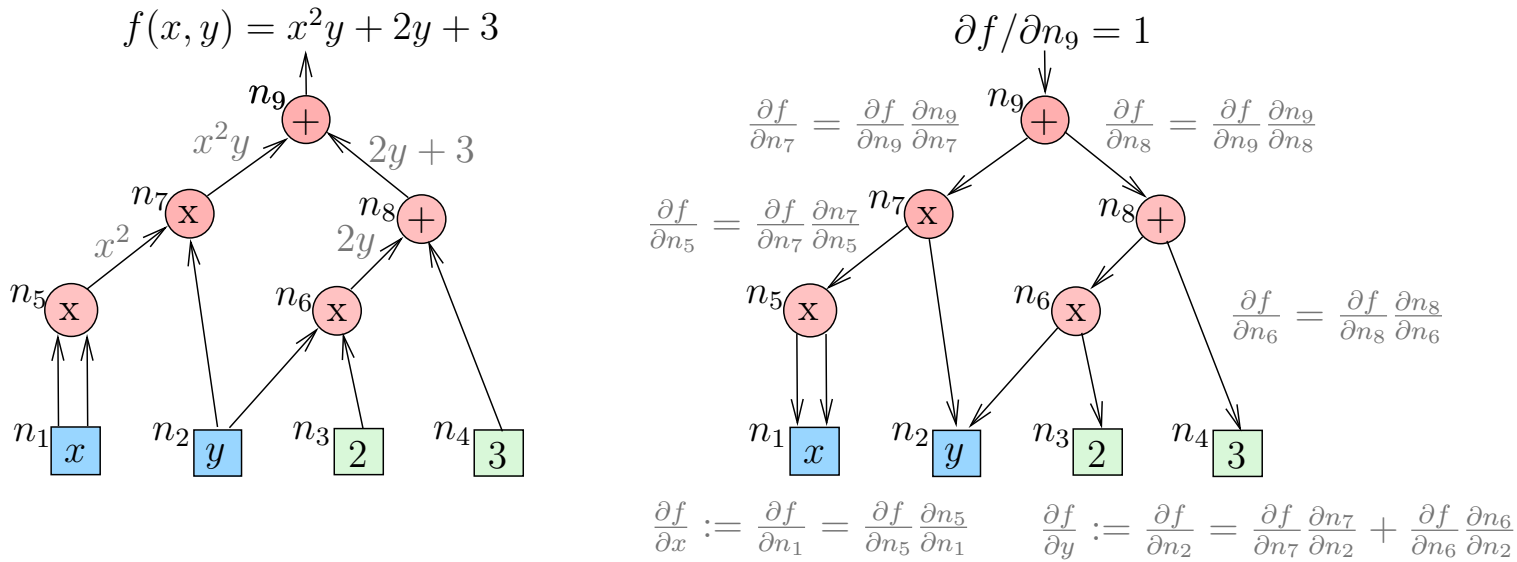

Figure: Forward evaluation (left) and reverse autodiff (right)

Forward pass to compute f at point $x = 4, y = -1$.
$n_5 = 16, \quad n_6 = -2, \quad n_7 = -16, \quad n_8 = 1, \quad n_9 = -15.$
Finally: $f(4, -1) = -15$.

Reverse pass to compute $\partial f/\partial x$ and $\partial f/\partial y$ at point $x = 4, y = -1$.
$\partial n_9/\partial n_7 = \partial n_9/\partial n_8 = 1, \quad \partial n_7/\partial n_5 = n_2 = -1, \quad \partial n_7/\partial n_2 = n_5 = 16,$
$\partial n_8/\partial n_6 = 1, \quad \partial n_5/\partial n_1 = 2n_1 = 8, \quad \partial n_6/\partial n_2 = n_3 = 2.$
Finally: $\partial f/\partial x = \ldots = -8$ and $\partial f/\partial y = \ldots = 18$.

Consider a computational graph with N_{in} inputs and N_{out} outputs.
We want to compute the derivatives of each output with respect to each input.
The characteristics of autodiff methods are summarized below

Method	Graph Traversals	Accuracy	Used in
Numerical	$N_{in} + 1$	Low	PDE grids
Symbolic	N/A	High	Mathematica
Fwd Autodiff	N_{in}	High	Stan
Rev Autodiff	$N_{out} + 1$	High	MXNet, TensorFlow, ...

- In Machine Learning we typically have many more input features (N_{in}) than outputs (N_{out}).
- The preferred method of most ML algorithms is reverse-mode autodiff.

Nonlinearity in ANN enters via non-linear activation functions.
Activation functions act element-wise on a vector $\mathbf{x}$ of size n.
Common activations are:

$$\text{relu}(\mathbf{x}) = [\max(x_1, 0), \cdots, \max(x_n, 0)]^T \tag{4}$$

$$\text{softplus}(\mathbf{x}) = [\log(1 + \exp(x_1)), \cdots, \log(1 + \exp(x_n))]^T \tag{5}$$

$$\text{sigmoid}(\mathbf{x}) = \left[\frac{1}{1 + \exp(-x_1)}, \cdots, \frac{1}{1 + \exp(-x_n)}\right]^T \tag{6}$$

$$\tanh(\mathbf{x}) = \left[\frac{1 - \exp(-2x_1)}{1 + \exp(-2x_1)}, \cdots, \frac{1 - \exp(-2x_n)}{1 + \exp(-2x_n)}\right]^T \tag{7}$$

$$\text{softmax}(\mathbf{x}) = \frac{1}{\sum_{i=1}^{n} \exp(x_i)} [\exp(x_1), \cdots, \exp(x_n)]^T \tag{8}$$

We encountered sigmoid and softmax in binary and multiclass logistic regression.

The simplest and most popular activation is the **Re**ctifier **L**inear **U**nit (ReLU). The softplus is its smoothed version.

- Unlike sigmoid and tanh, ReLU does not saturate for large x (zero gradients), but it is not differentiable at $x = 0$
- Softplus is a is smoothed version of ReLU, differentiable everywhere
- Other ReLU variants are
  1. Leaky ReLU with hyperparameter $a \geq 0$.
     It has non-zero gradient for $x \leq 0$

$$f(x) = \begin{cases} x & x > 0 \\ ax & x \leq 0 \end{cases} \tag{9}$$

  2. Exponential ReLU (ELU) with hyperparameter $a \geq 0$.

$$f(x) = \begin{cases} x & x > 0 \\ a(e^x - 1) & x \leq 0 \end{cases} \tag{10}$$

Frameworks and Interfaces:

ANNs are built from components (layers, optimizers, cells, losses, metrics ...). ANN frameworks are software libraries that provide re-usable components that are designed to fit together, and have been tested by many users.

We distinguish between **back-end** frameworks and **interface** libraries. Back-ends do the actual work. They are typically written in C/C++, they can execute on CPUs and GPUs, support common OS, and are high level programmable. Common ANN frameworks are

Framework	Languages	Creator
CNTK	C++, Python	Microsoft
H2O	Python, Java	H2O.ai
MXNet	C++, Python, R, Julia	DMLC, Amazon
TensorFlow	C++, Python	Google
Theano	Python, C	U. Montreal
Torch	C++, Lua	R. Collobert et al

Interface libraries are user friendly front-ends that can connect to different back-ends. Common interfaces are Keras, Gluon, Pytorch.

- In this class we will use **Keras** front-end with **TensorFlow** back-end.

Frameworks and Interfaces: Set-up (I)

TensorFlow supports Python 3.7 (since version 1.13), but Keras does not, yet. We need to create an environment based on Python 3.6 and install TensorFlow and Keras there. This is easily done from the Anaconda Prompt.

```
(base)> conda create -n py36 python=3.6 anaconda
(base)> conda activate py36
```

The new installation already contains the packages relevant for this class

```
numpy, scipy, pandas, matplotlib, sklearn, statsmodels
```

In addition we install the ANN packages as follows

```
(py36)> conda install -c conda-forge tensorflow
(py36)> conda install -c conda-forge keras
```

We test that we can invoke the main user apps that ship with Anaconda

```
(py36)> ipython
(py36)> spyder
(py36)> jupyter lab
```

Within each app we should be able to import all of the above packages.

NOTE: When an environment gets activated, all batch scripts in the folder `activate.d` get executed. On Windows, when Keras gets activated, it executes the script `activate.d/vs2015_compiler_vars.bat`. If the machine does not have a Visual Studio 2015 installation, the script may abort the shell. Simply rename the script suffix .bat to avoid it.

Validating the setup

```
(py36)> ipython
In [1]: import keras
In [2]: keras.__version__
Out[2]: '2.2.4'
In [3]: keras.backend.backend()
Out[3]: 'tensorflow'
In [4]: keras.backend.tensorflow_backend.tf.__version__
Out[4]: '1.10.0'
```

Exploring the toolset

```
In [5]: dir(keras)
Out[5]: ['Input', 'Model', 'Sequential', 'activations',
         'backend', 'datasets', 'layers', 'models',
         'metrics', 'optimizers', 'utils', ...]
In [6]: dir(keras.datasets)
Out[6]: ['boston_housing', 'cifar', 'fashion_mnist', 'mnist', 'reuters', ...]
In [7]: dir(keras.layers)
Out[7]: ['Activation', 'Dense', 'Dropout', ...]
In [8]: dir(keras.activations)
Out[8]: ['elu', 'relu', 'sigmoid', 'softmax', 'softplus', 'tanh',...]
In [9]: dir(keras.optimizers)
Out[9]: ['Adadelta', 'Adagrad', 'Adam', 'Adamax', 'SGD', 'RMSProp', ...]
In [10]: dir(keras.metrics)
Out[10]: ['binary_crossentropy', 'categorical_crossentropy', 'mae', 'mse', ...]
```

In Keras we use the standard workflow: **construct→compile→fit→evaluate**
Keras provides two interfaces (methods of constructing ANNs)

1. **Sequential**: using the class `keras.models.Sequential`
   - Layers are added to the Sequential class, either as parameters to the constructor, or by calling the `.add()` method
   - The final result is a sequential network, i.e. with one non-branching sequence of layers
2. **Functional**: using the class `keras.models.Model`
   - Sequences of layers are constructed by passing one layer to the next.
   - The sequences are passed to the Model constructor or the `.add()` method
   - This allows for network topologies with branching/merging sequences

With both interfaces, after the model has been constructed we can call the `.summary()` method to display its topology, or the `plot_model()` function to visualize it on an image file.

After construction, a model needs to be compiled by calling:

```
model.compile(optimizer=..., loss=..., metrics=...)
```

Fitting is done by calling: `model.fit(x_train, y_train, batch_size=..., epochs=...)`

Finally evaluation is done by calling: `model.evaluate(x_test, y_test)`

At this initial phase we focus on the following Keras core layers:

1. **Dense**: the workhorse layer for fully connected networks.
   - The only required constructor parameter is the number of units (neurons)
   - Every input is connected to every output
   - Each connection introduces a weight
   - Weights are grouped in "kernels" (matrix W) and biases (vector b)
   - Initialization and regularization of kernels and biases is done at construction time
   - Activation can set in the constructor or added later as a separate layer.
2. **Activation**: layer that applies the activation function on each component of the input
   - Returns same shape transformed output
   - The activation function can be specified by name or user supplied
3. **Dropout**: drops a fraction of the input observations
   - It is used to combat overfitting
4. **Reshape**: reshapes its input to the specified output shape

For details see the notebook **L10-ANN-Examples.ipynb**

Further Reading

1. Follow the steps in subsection "Set-up" to install and verify your environment
2. Read sections 5.1-5.3 in [BI] "Neural Networks".
   It has a nice exposition of the ANN math.
3. Optionally, read chapter 10 in [GA] "Introduction to Artificial Neural Networks". It contains details on TensorFlow internals.
4. The documentation of Keras is available online at keras.io

References

[BI] Bishop C.M. "Pattern Recognition and Machine Learning". Springer, 2006

[GA] Géron Aurélien. "Hands-On Machine Learning with Scikit-Learn and TensorFlow". O'Reilly, 2017