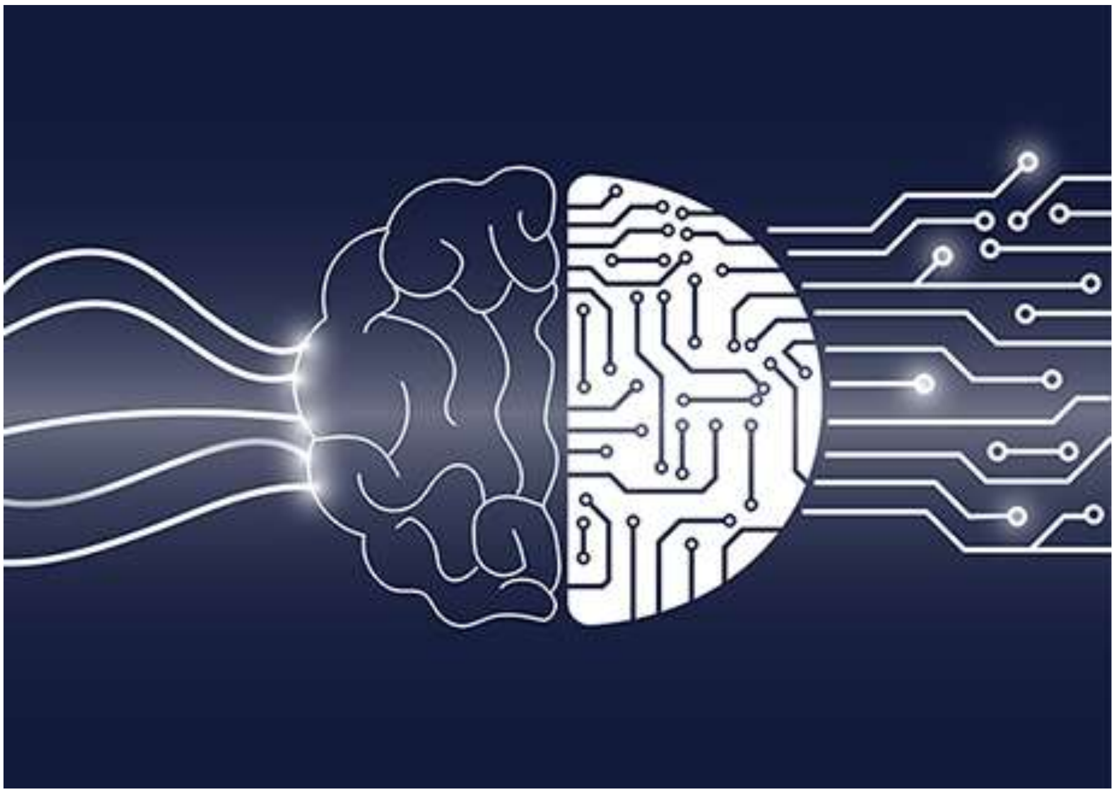
NATIONAL TECHNICAL UNIVESITY OF ATHENS

Civil Engineering Department
Laboratory of Structural Analysis and Antiseismic Research

Application of Deep Neural Networks in Engineering

Post-Graduate Thesis



Katsimpalis Emmanouil

Supervisor: Professor Vlasis Koumousis

Athens, June 2019

Katsimpalis E. G. (2019).
Application of Deep Neural Networks in Engineering
Post-Grad Thesis
Laboratory of Structural Analysis and Antiseismic Research, National Technical University of Athens, Greece

Table of Contents

NATIONAL TECHNICAL UNIVERSITY OF ATHENS
FACULTY OF CIVIL ENGINEERING
LABORATORY OF STRUCTURAL ANALYSIS AND ANTISEISMIC RESEARCH
POST-GRADUATE THESIS

Application of Deep Neural Networks in Engineering

Katsimpalis E. G. (supervised by Koumousis V.)

Abstract

In this post-graduate thesis, the aim was to explore the capabilities of deep learning and find applications in engineering problems. Deep learning is an exciting spin on the concept of machine learning and has been applied with great success in many fields including computer vision, speech recognition, natural language processing, audio recognition, social network filtering, etc. Most modern deep learning models are based on artificial neural networks. In this work, the most popular neural networks are presented, which include feedforward neural networks, convolutional neural networks (CNN) and recurrent neural networks (RNN). These approaches were applied in supervised learning tasks, such as classification and regression, using the “Python” library “Keras” to build, train and test various models. In the first chapter, the fundamental concepts of machine learning are presented , including loss functions, gradient based optimization and backpropagation. At the end of the chapter, we move from machine learning to deep learning with the introduction of the quintessential neural network, the feedforward neural network. Chapter 2 introduces convolutional neural networks, a class of deep neural networks that has produced impressive results on classification tasks such as image recognition. Afterwards, in chapter 3 recurrent neural networks are presented, which improve on feedforward neural networks by adding feedback and are one of the more popular approaches to regression tasks. Chapter 4 discusses ways to evaluate the performance of the models and minimize the generalization error, which is the error of the model on previously unseen data. Lastly, in chapter 5 the data preparation process is shown and in chapter 6 models are build and trained, using “Python” libraries like “Keras” , to tackle various classification and regression tasks.

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΠΟΛΙΤΙΚΩΝ ΜΗΧΑΝΙΚΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΣΤΑΤΙΚΗΣ ΚΑΙ ΑΝΤΙΣΕΙΣΜΙΚΩΝ ΕΡΕΥΝΩΝ
ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Εφαρμογές Νευρωνικών Δικτύων στη Μηχανική

Κατσίμπαλης Ε. Γ. (Επιβλέπων: Κουμούσης Β.)

Περίληψη

Αντικείμενο της παρούσας μεταπτυχιακής εργασίας αποτελεί η διερεύνηση των δυνατοτήτων της βαθιάς μάθησης στη μηχανική. Η βαθιά μάθηση αποτελεί μια συναρπαστική παραλλαγή της μηχανικής μάθησης και έχει εφαρμοστεί με μεγάλη επιτυχία σε πολλά πεδία της επιστήμης όπως είναι η τεχνητή όραση, η αναγνώριση ομιλίας, η επεξεργασία ξένων γλωσσών, η μουσική αναγνώριση, η εφαρμογή διάφορων φίλτρων σε κοινωνικά δίκτυα κλπ. Τα περισσότερα μοντέρνα μοντέλα βαθιάς μάθησης βασίζονται στα τεχνητά νευρωνικά δίκτυα. Στην παρούσα εργασία παρουσιάζονται τα πιο δημοφιλή νευρωνικά δίκτυα, τα οποία περιλαμβάνουν τα εμπρόσθια τροφοδοτούμενα νευρωνικά δίκτυα(feedforward neural networks) , συνελικτικά νευρωνικά δίκτυα (convolutional neural networks) και τα επαναλαμβανόμενα νευρωνικά δίκτυα (recurrent neural networks). Αυτού του είδους οι προσεγγίσεις εφαρμόστηκαν σε συνήθη προβλήματα μηχανικής μάθησης, όπως ταξινόμησης και παλινδρόμησης, χρησιμοποιώντας βιβλιοθήκες «Python» όπως την «Keras» για να κατασκευάσουμε, εκπαιδεύσουμε και ελέγξουμε διάφορα μοντέλα. Στο πρώτο κεφάλαιο, παρουσιάζονται οι βασικές αρχές της μηχανικής μάθησης, συμπεριλαμβανομένων των συναρτήσεων απώλειας, της βελτιστοποίησης βασισμένης στην ελαχιστοποίηση κλίσης και της οπισθοδιάδοσης. Στο τέλος του κεφαλαίου γίνεται μετάβαση από την μηχανική μάθηση στην βαθιά μάθηση με την εισαγωγή του ποιο αντιπροσωπευτικού νευρωνικού δικτύου, του εμπρόσθια τροφοδοτούμενου νευρωνικού δικτύου. Στο 2^{ο} κεφάλαιο παρουσιάζονται τα συνελικτικά νευρωνικά δίκτυα, μία κατηγορία νευρωνικών δικτύων με εντυπωσιακά αποτελέσματα σε εργασίες ταξινόμησης όπως είναι η αναγνώριση εικόνων. Στη συνέχεια συζητήθηκαν τα επαναλαμβανόμενα νευρωνικά δίκτυα στο 3^{ο} κεφάλαιο, τα οποία βελτιώνουν τα εμπρόσθια τροφοδοτούμενα νευρωνικά δίκτυα με την προσθήκη επανατροφοδότησης και είναι από τις ποιο δημοφιλείς προσεγγίσεις σε προβλήματα παλινδρόμησης. Στη συνέχεια στο κεφάλαιο 4 παρουσιάζονται τρόποι για την αξιολόγηση των επιδόσεων των μοντέλων και την ελαχιστοποίηση του λάθους γενίκευσης, το οποίο είναι η απώλεια του μοντέλου σε άγνωστα προηγουμένως δεδομένα. Τέλος, στο κεφάλαιο 5 παρουσιάζεται η διαδικασία προετοιμασίας δεδομένων και στο 6^{ο} κεφάλαιο κατασκευάζονται μοντέλα και εκπαιδεύονται με χρήση βιβλιοθηκών «Python» , κυρίως της βιβλιοθήκης «Keras», για την αντιμετώπιση διαφόρων προβλημάτων ταξινόμησης και παλινδρόμησης.

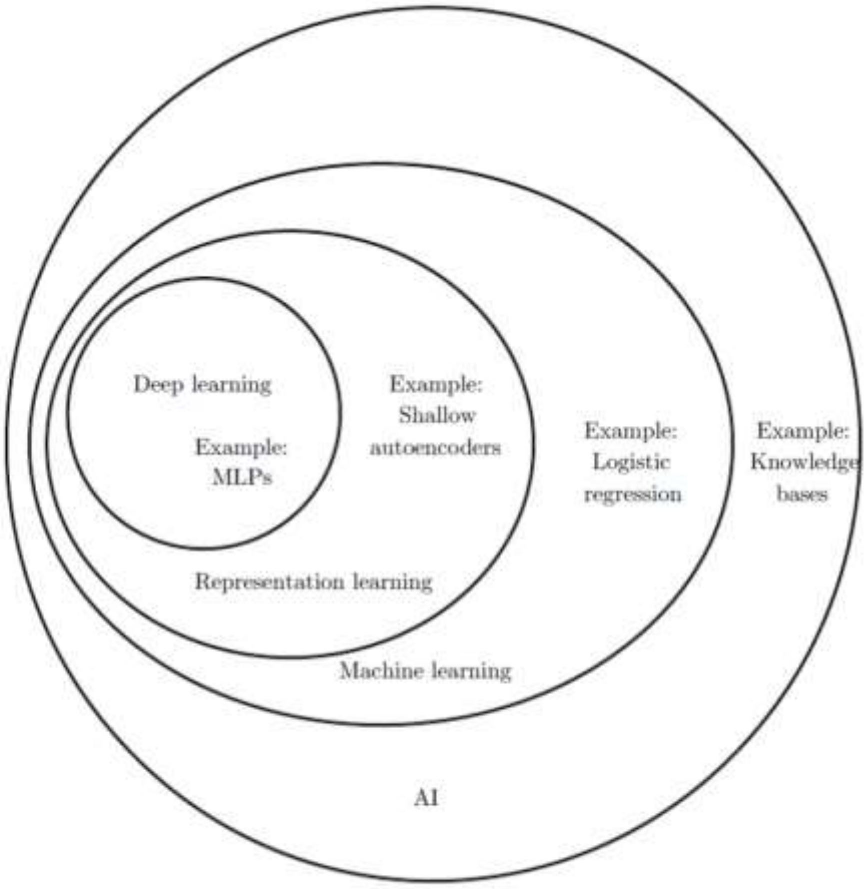
Acknowledgements

Throughout the writing of this thesis I have received a great deal of support and assistance. I would first like to thank my supervisor, Mr Koumousis Vlassis, professor at the National Technical University of Athens. He introduced me to optimization and machine learning through his courses in this postgraduate program. After the completion of the courses, he offered me the opportunity to dive deeper into the subject in this thesis and provided valuable guidance and encouragement to experiment with new approaches.
In addition, I would like to thank my parents for their wise counsel and for always being there for me. Finally, there are my friends, who were of great support in deliberating over our problems and findings, as well as providing happy distraction to rest my mind outside of my research.

1 Deep Learning

1.1 Machine learning

The focus of this work will be on using **deep neural networks in supervised learning tasks**. However, deep learning is a specific kind of machine learning. In order to understand deep learning well, one must have a solid understanding of the basic principles of machine learning.



Mitchell (1997) provides the definition “A computer program is said to learn from experience *E* with respect to some class of tasks *T* and performance measure *P* , if its performance at tasks in *T* , as measured by *P* , improves with experience *E*.”

1.1.1 Task

Many different problems can be tackled with machine learning. Some of the most common machine learning tasks include the following:

- Classification: In this type of task, the computer program is asked to specify which of *k* categories some input belongs to.
- Classification with missing inputs: Classification becomes more challenging if the computer program is not guaranteed that every measurement in its input vector will always be provided. In order to solve the classification task, the learning algorithm only has to define a single function mapping from a vector input to a categorical output. When some of the inputs maybe missing, rather than providing a single classification function, the learning algorithm must learn a set of functions

- Regression: In this type of task, the computer program is asked to predict a numerical value given some input.
- Transcription: In this type of task, the machine learning system is asked to observe a relatively unstructured representation of some kind of data and transcribe it into discrete, textual form. For example, in optical character recognition, the computer program is shown a photograph containing an image of text and is asked to return this text in the form of a sequence of characters
- Machine translation: In a machine translation task, the input already consists of a sequence of symbols in some language, and the computer program must convert this into a sequence of symbols in another language.
- Structured output: Structured output tasks involve any task where the output is a vector (or other data structure containing multiple values) with important relationships between the different elements.
- Anomaly detection: In this type of task, the computer program sifts through a set of events or objects, and flags some of them as being unusual or atypical
- Synthesis and sampling: In this type of task, the machine learning algorithm is asked to generate new examples that are similar to those in the training data
- Imputation of missing values: In this type of task, the machine learning algorithm is given a new example $\boldsymbol{x} \in R^n$, but with some entries x_i of $\boldsymbol{x}$ missing. The algorithm must provide a prediction of the values of the missing entries.
- Denoising: In this type of task, the machine learning algorithm is given in input a corrupted example $\widehat{\boldsymbol{x}} \in R^n$ obtained by an unknown corruption process from a clean example $\boldsymbol{x} \in R^n$.

1.1.2 Experience

Machine learning algorithms can be broadly categorized as unsupervised or supervised by what kind of experience they are allowed to have during the learning process. However, recently two more types of learning have been introduced, semi-supervised and reinforced learning.

Supervised learning algorithms experience a dataset containing features, but each example is also associated with a label or target. For example, the Iris dataset is annotated with the species of each iris plant. A supervised learning algorithm can study the Iris dataset and learn to classify iris plants into three different species based on their measurements. The majority of practical machine learning uses supervised learning. The most popular supervised algorithms focus on **classification** and **regression.**

Unsupervised learning algorithms experience a dataset containing many features, then learn useful properties of the structure of this dataset. In the context of deep learning, we usually want to learn the entire probability distribution that generated a dataset, whether explicitly as in density estimation or implicitly for tasks like synthesis or denoising. Some other unsupervised learning algorithms perform other roles, like clustering, which consists of dividing the dataset into clusters of similar examples. The most popular unsupervised algorithms focus on **clustering** and **association**.

Semi-supervised learning is a class of machine learning tasks and techniques that also make use of unlabeled data for training – typically a small amount of labeled data with a large amount of unlabeled data. Semi-supervised learning falls between unsupervised learning (without any labeled training data) and supervised learning (with completely labeled training data).

Reinforcement learning is an area of machine learning concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward. It differs from supervised learning in that labelled input/output pairs need not be presented, and sub-optimal actions need not be explicitly corrected. Instead the focus is finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge).

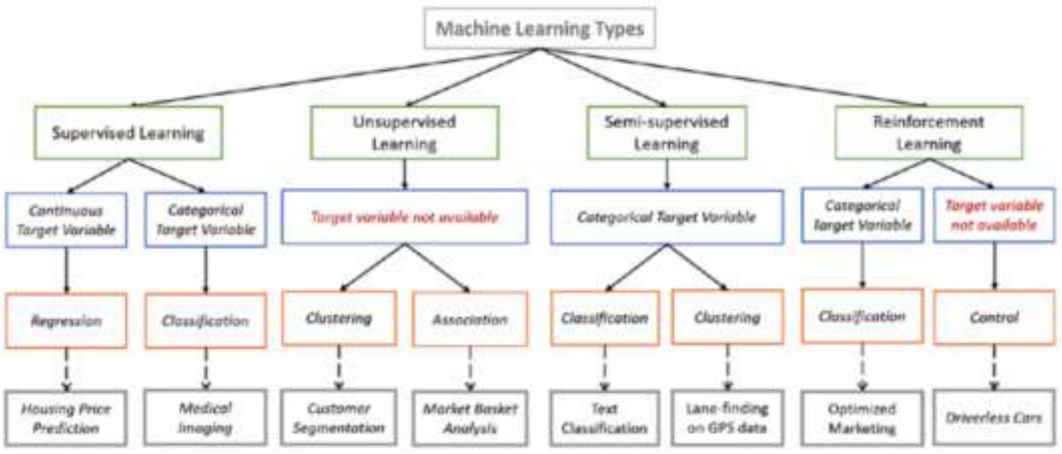


image from en.proft.me

1.2 Essential Machine learning Concepts

1.2.1 Cost Functions

An important aspect of the design of a machine learning algorithm is the choice of the cost function. Fortunately, the cost functions for neural networks are more or less the same as those for other parametric models, such as linear models.
In machine learning, cost functions are used to estimate how badly models are performing. Put simply, a cost function is a measure of how wrong the model is in terms of its ability to estimate the relationship between x and y. This is typically expressed as a difference or distance between the predicted value and the actual value. The cost function (you may also see this referred to as loss or error.) can be estimated by iteratively running the model to compare estimated predictions against "ground truth" — the known values of y. The objective of a ML model, therefore, is to find parameters, weights or a structure that minimizes the cost function.
There exist many cost functions that are suited to various machine learning problems. The ones presented bellow are those which will be used in later chapters on classification problems.

1.2.1.1 Mean Squared Error (MSE)

MSE is perhaps the most simple and common metric for regression evaluation, but also probably the least useful. Assuming we have a model that predicts n values of variable $\bar{y}_i$, while the target outputs are y_i. The mean squared error is calculated as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{y}_i)$$

This cost function and some of its variants are useful for simple machine learning models. However, if we make a single very bad prediction, the squaring will make the error even worse and it may skew the metric towards overestimating the model's error. That is a particularly problematic behaviour if we have noisy data (that is, data that for whatever reason is not entirely reliable) — even a "perfect" model may have a high MSE in that situation, so it becomes hard to judge how well the model is performing. On the other hand, if all the errors are small, or rather, smaller than 1, than the opposite effect is felt: we may underestimate the model's error.

1.2.1.2 Cross-Entropy

Cross-entropy is commonly used to quantify the difference between two probability distributions. Usually the "true" distribution (the one that your machine learning algorithm is trying to match) is expressed in terms of a one-hot distribution.

For example, suppose for a specific training instance, the label is B (out of the possible labels A, B, and C). The one-hot distribution for this training instance is therefore:

```
Pr(Class A)  Pr(Class B)  Pr(Class C)
        0.0          1.0          0.0
```

You can interpret the above "true" distribution to mean that the training instance has 0% probability of being class A, 100% probability of being class B, and 0% probability of being class C.

Now, suppose your machine learning algorithm predicts the following probability distribution:

```
Pr(Class A)  Pr(Class B)  Pr(Class C)
      0.228        0.619        0.153
```

The cross-entropy loss determines how close the predicted probabilities are to the true distribution using this formula:

$$H(p,q) = -\sum_{x} p(x)\log(q(x))$$

Where $p(x)$ is the wanted probability, and $q(x)$ the actual probability. The sum is over the three classes A, B, and C. In this example:

```
H = - (0.0*ln(0.228) + 1.0*ln(0.619) + 0.0*ln(0.153)) = 0.479
```

So that is how "wrong" or "far away" your prediction is from the true distribution.

1.2.2 Gradient-based Optimization

Currently, the most popular approach to machine learning is based on minimizing the gradient of a cost function. Especially in deep learning, the training algorithm is usually based on using the gradient to descend the cost function in one way or another.

From "Deep Learning" by Ian Goodfellow, Yoshua Benshio, Aaron Courville
"The largest difference between the linear models and neural networks is that the nonlinearity of a neural network causes most interesting loss functions to become non-convex. This means that neural networks are usually trained by using iterative, gradient-based optimizers that merely drive the cost function to a very low value, rather than the

linear equation solvers used to train linear regression models or the convex optimization algorithms with global convergence guarantees used to train logistic regression or SVMs. Convex optimization converges starting from any initial parameters (in theory—in practice it is very robust but can encounter numerical problems). Stochastic gradient descent applied to non-convex loss functions has no such convergence guarantee, and is sensitive to the values of the initial parameters."

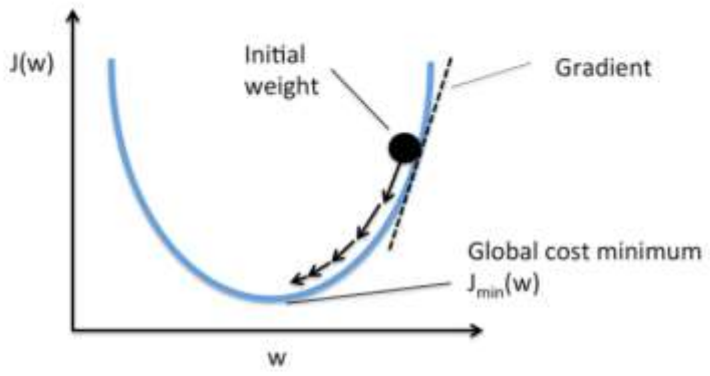
1.2.2.1 Gradient Descent method

Using the Gradient Decent (GD) optimization algorithm, the weights are updated incrementally after each epoch (= pass over the training dataset). The magnitude and direction of the weight update is computed by taking a step in the opposite direction of the cost gradient.

$$\Delta w_j = -n\frac{dJ}{dw_j}$$

where n is the learning rate. The weights are then updated after each epoch via the following update rule:

$$\boldsymbol{w} = \boldsymbol{w} + \Delta\boldsymbol{w}$$



1.2.2.2 Stochastic Gradient Descent

In GD optimization, we compute the cost gradient based on the complete training set; hence, we sometimes also call it batch GD. In case of very large datasets, the use of GD can be quite costly since we are only taking a single step for one pass over the training set. Thus, the larger the training set, the slower our algorithm updates the weights and the longer it may take until it converges to the global cost minimum (note that the SSE cost function is convex).

In Stochastic Gradient Descent (SGD; sometimes also referred to as iterative or on-line GD), we don't accumulate the weight updates as we've seen above for GD:

- For one or more epochs
    - For each weight j

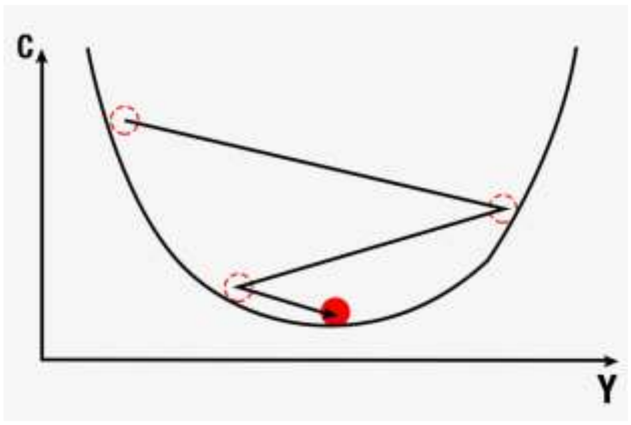
$$w_j = w_j - n\frac{dJ}{dw_j}$$

Instead, we update the weights after each training sample:

- For one or more epochs
    - For each training sample i
        - For each weight j

$$w_j = w_j - n\frac{dJ}{dw_j}$$

Here, the term "stochastic" comes from the fact that the gradient based on a single training sample is a "stochastic approximation" of the "true" cost gradient. Due to its stochastic nature, the path towards the global cost minimum is not "direct" as in GD, but may go "zig-zag" if we are visualizing the cost surface in a 2D space.



However, it has been shown that SGD almost surely converges to the global cost minimum if the cost function is convex (or pseudo-convex). Furthermore, there are different tricks to improve the GD-based learning, for example:

-An adaptive learning rate η choosing a decrease constant d (decay) that shrinks the learning rate over time:

$$n(t+1) = \frac{n(t)}{(1+t*d)}$$

- Momentum learning by adding a factor of the previous gradient to the weight update for faster updates:

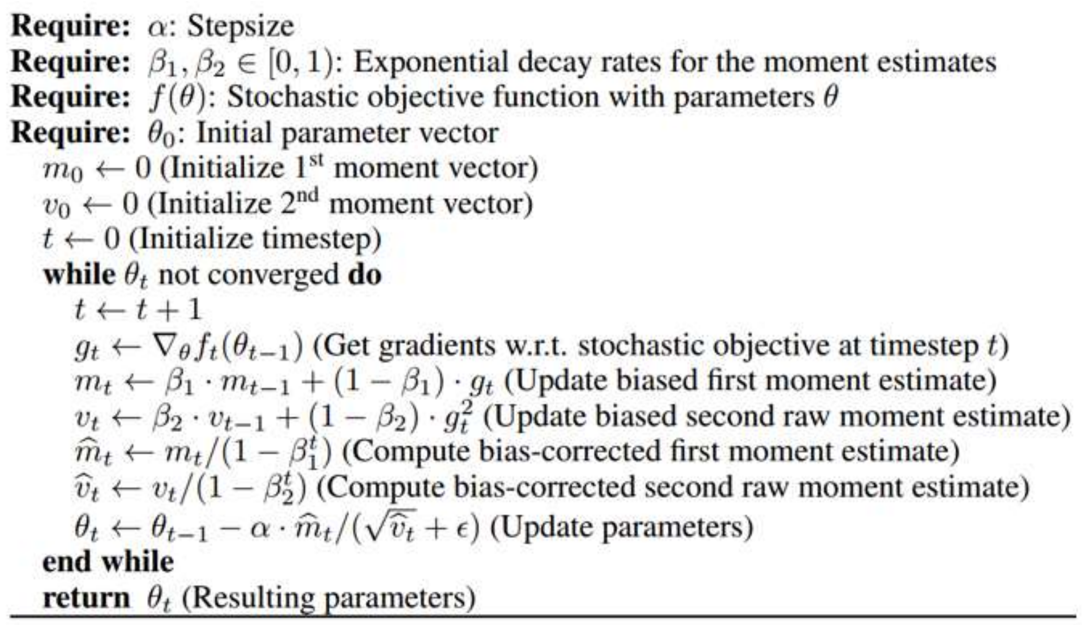
$$\Delta \boldsymbol{w}_{t+1} = n\nabla \boldsymbol{J}(\boldsymbol{w}_{t+1}) + a\Delta \boldsymbol{w}_t$$

1.2.2.3 Adam Optimization Algorithm

Adam is an optimization algorithm that can used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data.

It was presented by Diederik Kingma from OpenAI and Jimmy Ba from the University of Toronto in their 2015 ICLR paper (poster) titled “Adam: A Method for Stochastic Optimization“. The name Adam is derived from adaptive moment estimation.

The image bellow depicts the procedure as presented in the original paper.



```
**Require:** $\alpha$: Stepsize
**Require:** $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
**Require:** $f(\theta)$: Stochastic objective function with parameters $\theta$
**Require:** $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  **while** $\theta_t$ not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t/(1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t/(1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t/(\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  **end while**
  **return** $\theta_t$ (Resulting parameters)
```

Image from original paper

1.2.3 Back-propagation

From “Deep Learning” by Ian Goodfellow, Yoshua Benshio, Aaron Courville
“When we use a neural network to accept an input x and produce an output $\tilde{y}$, information flows forward through the network. The inputs x provide the initial information that then propagates up to the hidden units at each layer and finally produces $\tilde{y}$,. This is called forward propagation. During training, forward propagation can continue onward until it produces a scalar cost $J(\theta)$. The back-propagation algorithm (Rumelhart et al., 1986a), often simply called backprop, allows the information from the cost to then flow backwards through the network, in order to compute the gradient.
Computing an analytical expression for the gradient is straightforward, but numerically evaluating such an expression can be computationally expensive. The back-propagation algorithm does so using a simple and inexpensive procedure.
The term back-propagation is often misunderstood as meaning the whole learning algorithm for multi-layer neural networks. Actually, back-propagation refers only to the method for computing the gradient, while another algorithm, such as stochastic gradient descent, is used to perform learning using this gradient.
Furthermore, back-propagation is often misunderstood as being specific to multilayer neural networks, but in principle it can compute derivatives of any function (for some functions, the correct response is to report that the derivative of the function is undefined).”
In learning algorithms, the gradient we most often require is the gradient of the cost function with respect to the parameters, $\nabla_\theta J(\theta)$.

1.2.3.1 Chain Rule of Calculus

The chain rule of calculus (not to be confused with the chain rule of probability) is used to compute the derivatives of functions formed by composing other functions whose derivatives are known. Back-propagation is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient.

Let x be a real number, and let f and g both be functions mapping from a real number to a real number. Suppose that *y = g(x)* and *z = f (g(x)) = f (y).* Then the chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

We can generalize this beyond the scalar case. Suppose that $\boldsymbol{x} \in R^m$, $\boldsymbol{y} \in R^n$, *g maps from* R^m to R^n, and *f* maps from R^n to R. If *y = g(x)* and *z = f(y)*, then

$$\frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j}\frac{dy_j}{dx_i}$$

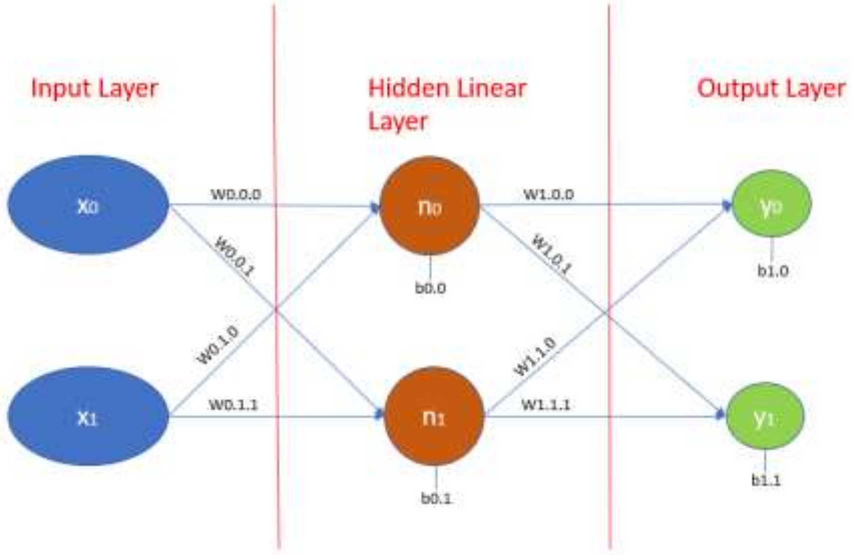
In vector notation, this may be equivalently written as

$$\nabla \boldsymbol{x}^z = \left(\frac{d\boldsymbol{y}}{d\boldsymbol{x}}\right)^T \nabla \boldsymbol{y}^z$$

where $\frac{dy}{dx}$ is the $n \times m$ Jacobian matrix of g.

1.2.4 Numerical example of supervised machine learning training

In order to get a better understanding of how a machine learning algorithm uses back-propagation to minimise the gradient and learn, a simple numerical example is presented. In the image bellow a simple neural network is depicted. The network has an input layer with 2 values, a hidden layer with 2 neurons and an output layer with 2 outputs.



Hidden Layer (Linear)

$n_0 = W_{0.0.0} * x_0 + W_{0.1.0} * x_1 + b_{0.0}$

$n_1 = W_{0.1.0} * x_0 + W_{0.1.0} * x_1 + b_{0.1}$

Output layer (Linear)

$y_0 = W_{1.0.0} * n_0 + W_{1.1.0} * n_1 + b_{1.0}$

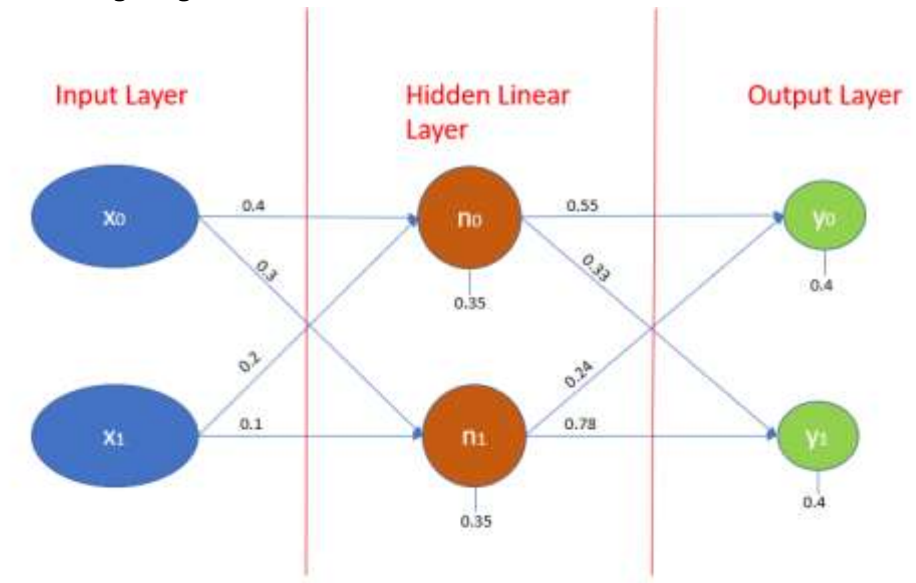
$y_0 = W_{1.0.0} * n_0 + W_{1.1.0} * n_1 + b_{1.1}$

As training data we have one sample with x_0=0.55 and x_1=0.67 as input and target output be y_{0_true}=0.01 and y_{1_true}=0.99.

Input X	x_0	x_1
Sample1	0.55	0.67

True Y	y_0	y_1
Sample1	0.01	0.99

Step 1: Initializing weights and biases



Step 2 Forward propagation
We solve the network given the input.

Hidden layer:
$n_0=0.4*0.55 + 0.2*0.67 + 0.35 = 0.704$
$n_1=0.3*0.55 + 0.1*0.67 + 0.35 = 0.582$

Output layer:
$y_{0\_est}=0.55*0.704 + 0.24*0.582 + 0.4 = 1.6092$
$y_{1\_est}=0.33*0.704 + 0.78*0.582 + 0.4 = 1.08628$

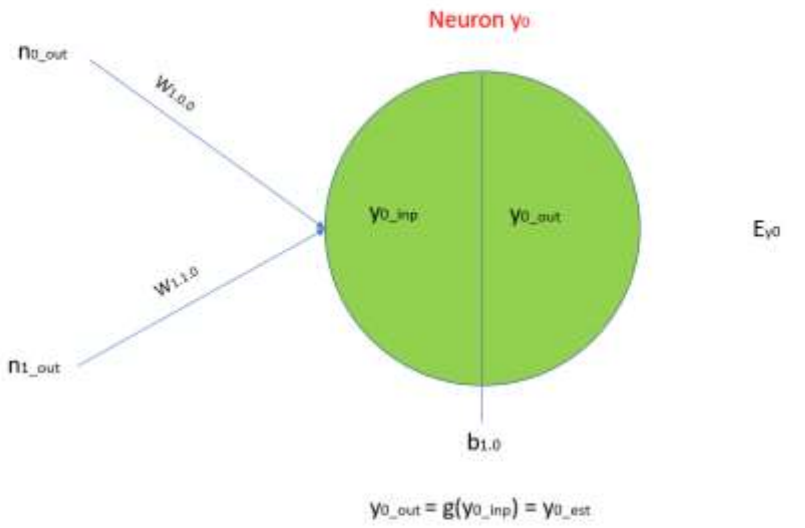
Step 3 Calculating the Total Error
For this example we will use the mean squared error (MSE) as a cost function.

$$\mathrm{E}_{total} = \sum \frac{1}{2}(true - estimated)^2$$

$$\mathrm{E}_{total} = \mathrm{E}_{y0} + \mathrm{E}_{y1} = \frac{1}{2}\left(y_{0\_true} - y_{0\_est}\right)^2 + \frac{1}{2}\left(y_{1\_true} - y_{1\_est}\right)^2 = 1.29315 + 0.00463$$
E_{total} =1.2978

Step 4 Backward propagation for output layer
Chain rule for weight W1.0.0:

$$\frac{dE_{total}}{dW_{1.0.0}} = \frac{dE_{total}}{dy_{0\_out}} * \frac{dy_{0\_out}}{dy_{0\_inp}} * \frac{dy_{0\_inp}}{dW_{1.0.0}}$$



g is an activation function, which is a method of expressing non-linearity. On the next chapter, when neural network algorithms like deep feedforward algorithm are introduced, activation functions will be discussed in length. For this example, since we assumed linear layers, y_{0_out}=y_{0_inp}.

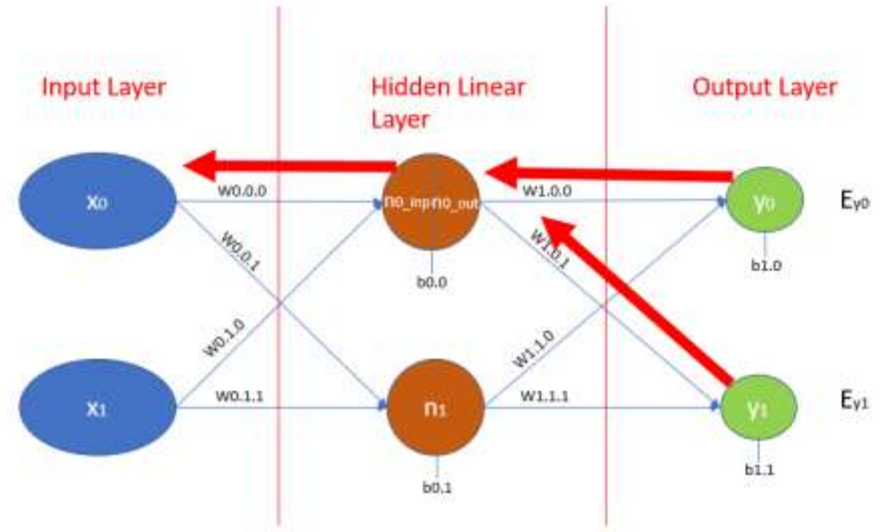
- $\frac{dE_{total}}{dy_{0\_out}} = -2\frac{1}{2}\left(y_{0\_true} - y_{0\_est}\right)^{2-1} = 1.5992$
- $\frac{dy_{0\_out}}{dy_{0\_inp}} = 1$ (since output layer is linear)
- $\frac{dy_{0\_inp}}{dW_{1.0.0}} = n_{0\_out} = 0.704$

Result:
$\frac{dE_{total}}{dW_{1.0.0}} = +1.1258$

Gradient descent

$$\mathrm{W}_{1.0.0}^{(1)} = \mathrm{W}_{1.0.0}^{(0)} - \text{ learning\_rate} * \frac{dE_{total}}{dW_{1.0.0}}$$
$$\mathrm{W}_{1.0.0}^{(1)} = 0.55 - 0.5 * (+1.1258) = 0.0129$$

Step 6 Backward propagation for hidden layer



Chain rule for weight $W_{0.0.0}$:

$$\frac{dE_{total}}{dW_{0.0.0}} = \frac{dE_{total}}{dn_{0\_out}} * \frac{dn_{0\_out}}{dn_{0\_inp}} * \frac{dn_{0\_inp}}{dW_{0.0.0}}$$

It is important to note that for the calculation of the weights on a layer the original values of weights at the start of the iterations are used, not the updated. Ex. $W_{1.0.0}^{(0)}$ will be used, not $W_{1.0.0}^{(1)}$ that was calculated in the previous step.

- $\frac{dE_{total}}{dn_{0\_out}} = \frac{dE_{y0}}{dn_{0\_out}} + \frac{dE_{y1}}{dn_{0\_out}}$
  - $\frac{dE_{y0}}{dn_{0\_out}} = \frac{dE_{y0}}{dy_{0\_inp}} * \frac{dy_{0\_inp}}{dn_{0\_out}}$
    - $\frac{dE_{y0}}{dy_{0\_inp}} = \frac{dE_{y0}}{dy_{0\_out}} * \frac{dy_{0\_out}}{dy_{0\_inp}} = -2\frac{1}{2}(y_{0\_true} - y_{0\_est})^{2-1} * 1 = 1.5992$
    - $\frac{dy_{0\_inp}}{dn_{0\_out}} = \frac{d(W_{1.0.0}*n_{0\_out}+W_{1.1.0}*n_{1\_out}+b_2)}{dn_{0\_out}} = W_{1.1.0}^{(0)} = 0.55$
  - $\frac{dE_{y1}}{dn_{0\_out}} = \frac{dE_{y1}}{dy_{1\_inp}} * \frac{dy_{1\_inp}}{dn_{0\_out}}$
    - $\frac{dE_{y1}}{dy_{1\_inp}} = \frac{dE_{y1}}{dy_{1\_out}} * \frac{dy_{1_{out}}}{dy_{1_{inp}}} = \left(-2\frac{1}{2}(y_{1_{true}} - y_{1_{est}})^{2-1}\right) * (1) = 0.09628$
    - $\frac{dy_{1\_inp}}{dn_{0\_out}} = \frac{d(W_{1.0.1}*n_{0\_out}+W_{1.1.1}*n_{1\_out}+b_2)}{dn_{0\_out}} = W_{1.0.1}^{(0)} = 0.33$

$$\frac{dE_{total}}{dn_{0_{out}}} = 1.5992 * 0.55 + 0.09628 * 0.33 = 0.91133$$

- $\frac{dn_{0_{out}}}{dn_{0_{inp}}} = 1$ (since hidden layer is linear)

- $\frac{dn_{0_{inp}}}{dW_{0.0.0}} = x_0 = 0.55$

Result:
$\frac{dE_{total}}{dW_{0.0.0}} = +0.5012$

Gradient descent

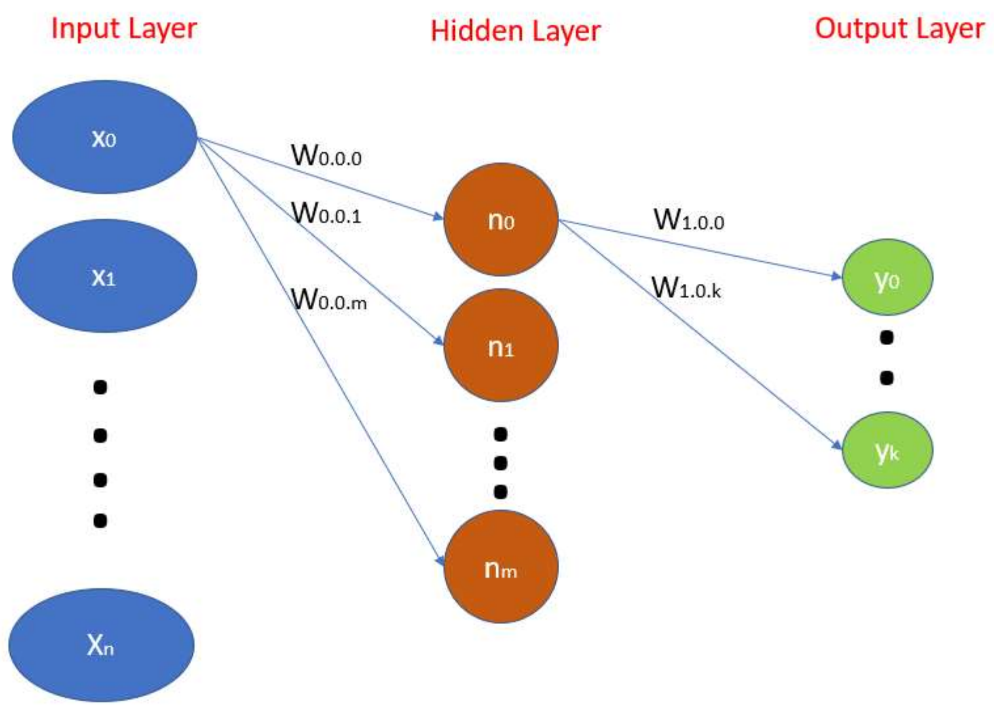
$$\mathrm{W}_{0.0.0}^{(1)} = \mathrm{W}_{0.0.0}^{(0)} - \text{ learning\_rate} * \frac{dE_{total}}{dW_{1.0.0}}$$
$$\mathrm{W}_{0.0.0}^{(1)} = 0.55 - \ 0.5 * (+0.5012) = 0.2994$$

1.3 Deep Feedforward Networks

So far, while we have introduced the concept of depth in the simple numerical example, we haven't discussed another important aspect of neural networks, which refers to non-linearity. In this section, feed forward networks will be presented.

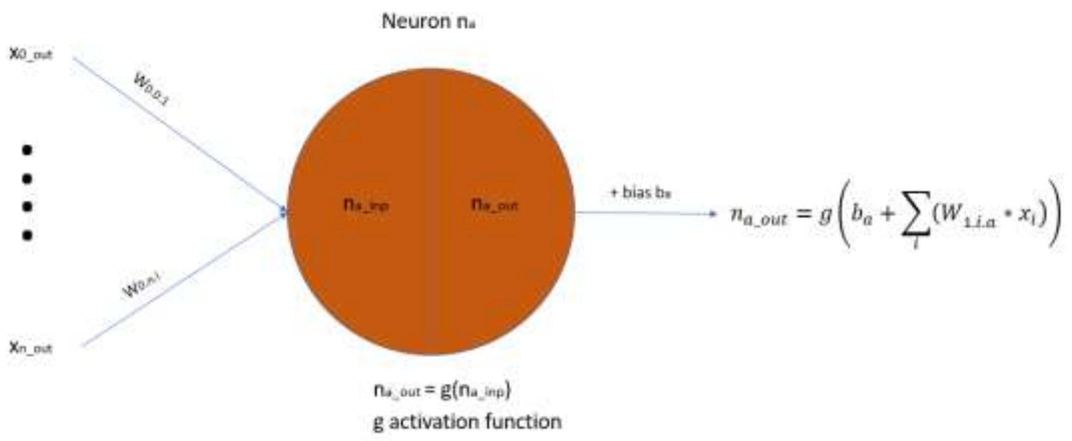
From "Deep Learning" by Ian Goodfellow, Yoshua Benshio, Aaron Courville
"Deep feedforward networks, also often called feedforward neural networks, or multilayer perceptrons (MLPs), are the quintessential deep learning models. The goal of a feedforward network is to approximate some function f^*. For example, for a classifier, $y = f^*(x)$ maps an input x to a category y. A feedforward network defines a mapping $y = f(x;\theta)$ and learns the value of the parameters θ that result in the best function approximation.
These models are called feedforward because information flows through the function being evaluated from x, through the intermediate computations used to define f , and finally to the output y. There are no feedback connections in which outputs of the model are fed back into itself. When feedforward neural networks are extended to include feedback connections, they are called recurrent neural networks.
Feedforward neural networks are called networks because they are typically represented by composing together many different functions. The model is associated with a directed acyclic graph describing how the functions are composed together. For example, we might have three functions $f^{(1)}$, $f^{(2)}$ and $f^{(3)}$ connected in a chain, to form $f(x) = f^{(3)}\left(f^{(2)}\left(f^{(1)}(x)\right)\right)$. These chain structures are the most commonly used structures of neural networks. In this case, $f^{(1)}$ is called the **first layer** of the network, $f^{(2)}$ is called the **second layer**, and so on. The overall length of the chain gives the **depth** of the model. It is from this terminology that the name "deep learning" arises. The final layer of a feedforward network is called the **output layer**."

Given below is an example of a feedforward Neural Network. It is a directed acyclic Graph which means that there are no feedback connections or loops in the network. It has an input layer, an output layer, and a hidden layer. In general, there can be multiple hidden layers. Each node in the layer is a Neuron, which can be thought of as the basic processing unit of a Neural Network.



$W_{l.i.j}$ where l is layer number, i is the starting neurons number and j is the destination neurons number

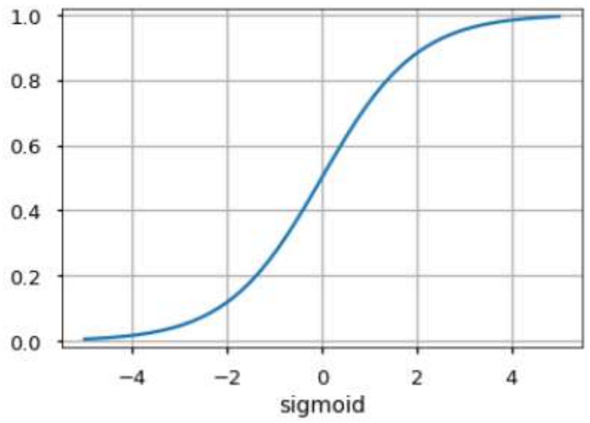
What is a Neuron?



As seen above, It works in two steps – It calculates the weighted sum of its inputs and then applies an activation function to normalize the sum. The activation functions can be linear or nonlinear. In addition, there are weights associated with each input of a neuron. These are the parameters, which the network has to learn during the training phase.
The activation function is used as a decision making body at the output of a neuron. The neuron learns Linear or Non-linear decision boundaries based on the activation function. It also has a normalizing effect on the neuron output, which prevents the output of neurons after several layers to become very large, due to the cascading effect. The most widely used activation functions are:

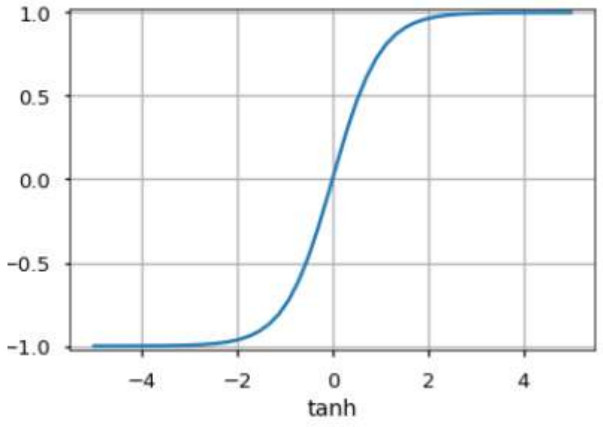
- **Sigmoid**

$$n_{a\_out} = \frac{1}{1 + e^{-n_{a\_inp}}}$$



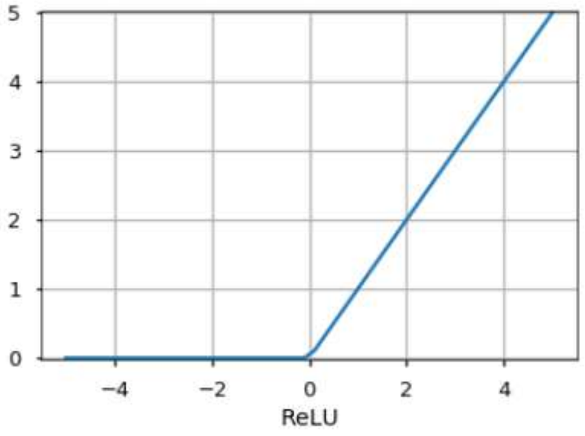
- **Tanh**

$$n_{a\_out} = \tanh(n_{a\_inp})$$



- **Rectified Linear Units (ReLU)**

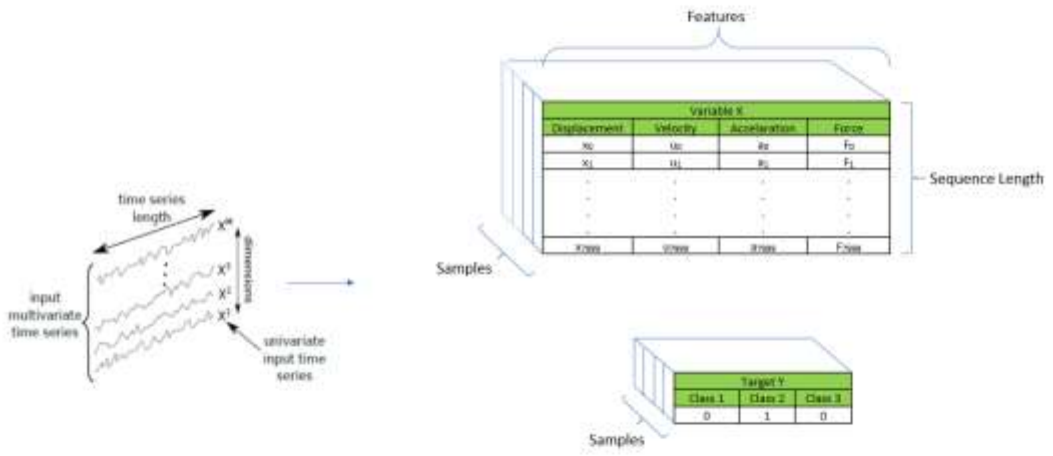
$$n_{a\_out} = max[0, n_{a\_inp}]$$



1.3.1 Example

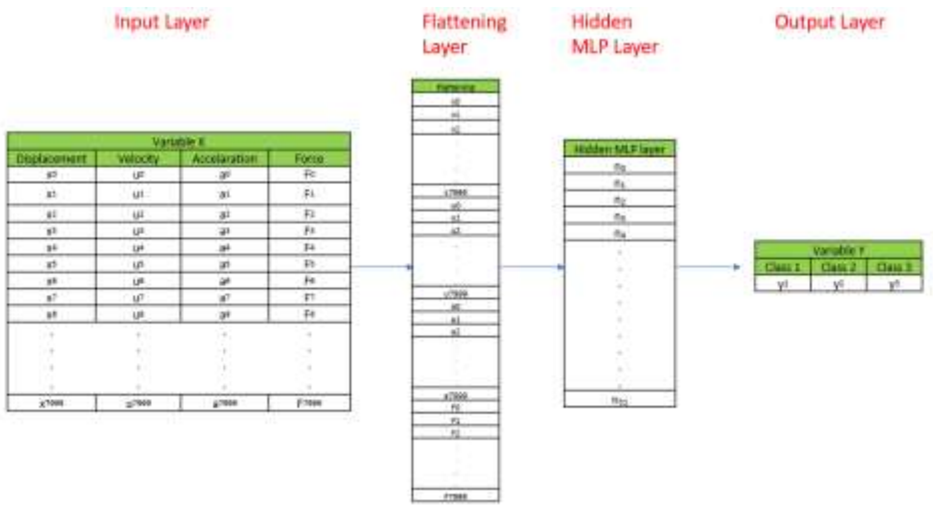
In order to get a better understanding of how multi layered perceptrons work, a simple example will be presented.
The task of the algorithm in this example is to identify the damping percentage of a SDOF oscillator subjected to ground motions. As input 4 features will be used, displacement,

velocity, acceleration and force, with values measured every dt=0.05 seconds over 40 seconds. This leads to a sequence length of 32000. As output there are three classes: class 1 for undamped, class 2 for damping 20% and class 3 for damping 40%.
First the input data are gathered in a matrix with dimensions (samples, sequence length, features). As target output we define a matrix with three columns, where we have a value of 1 in the column that corresponds to the correct category and 0 in the other columns for each row-sample. This is called one-hot distribution.

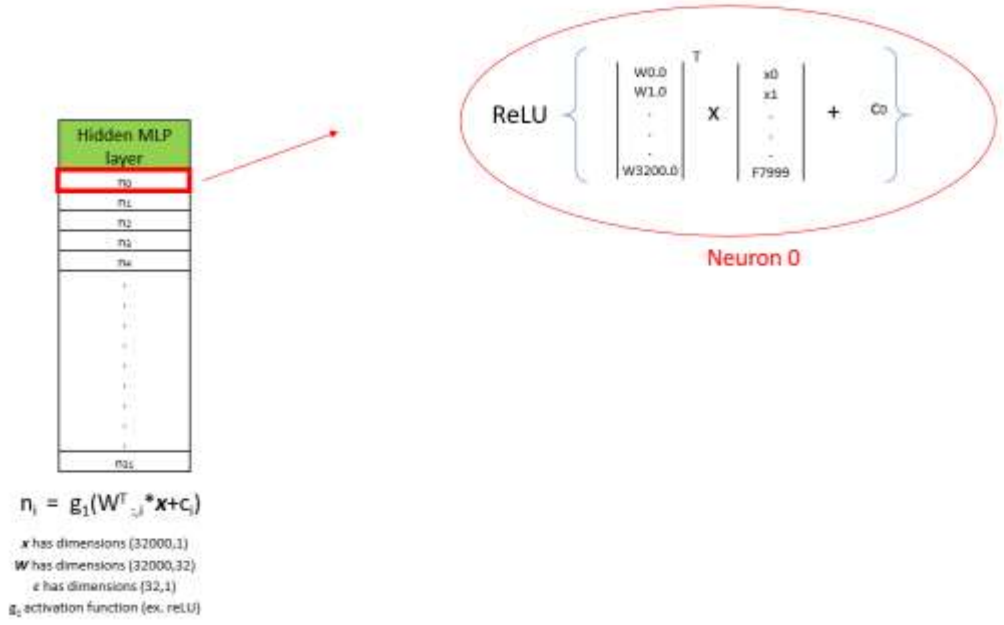


For this example a simple MLP layer with 32 neurons and ReLU activation function will be used. However since MLP layers can only have as input a 1 dimensional matrix, first we will flatten the data. So using stochastic gradient descent the following model will be used:

- For one or more epochs
    - For each training sample i



The learning process using backpropagation was presented in the previous chapter. But in this example we have non-linear layers. A visualization of each neuron is presented bellow.


Hidden MLP layer
n0
n1
n2
n3
n4
n31
$n_i = g_1(W^T_{:,i} * \boldsymbol{x} + c_i)$
x has dimensions (32000,1)
W has dimensions (32000,32)
c has dimensions (32,1)
g1 activation function (ex. reLU)
ReLU
W0.0
W1.0
W3200.0
T
x
x0
x1
F7999
+
C0
Neuron 0

From the above illustration, it is clear that the algorithm must learn 32000*32=1024000 weight parameters plus 32 bias parameters. This leads to 1024032 parameters for the hidden layer. For the output layer following the same logic we get, 32*3 weight parameters plus 3 biases, 99 learning parameters.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_2 (Flatten)          (None, 32000)             0
_________________________________________________________________
dense_2 (Dense)              (None, 32)                1024032
_________________________________________________________________
dense_3 (Dense)              (None, 3)                 99
=================================================================
Total params: 1,024,131
Trainable params: 1,024,131
Non-trainable params: 0
_________________________________________________________________
```

2 Convolutional Neural Networks

From “Deep Learning” by Ian Goodfellow, Yoshua Benshio, Aaron Courville
“Suppose we are tracking the location of a spaceship with a laser sensor. Our laser sensor provides a single output x(t), the position of the spaceship at time t. Both x and t are real-valued, i.e., we can get a different reading from the laser sensor at any instant in time.
Now suppose that our laser sensor is somewhat noisy. To obtain a less noisy estimate of the spaceship’s position, we would like to average together several measurements. Of course, more recent measurements are more relevant, so we will want this to be a weighted average that gives more weight to recent measurements. We can do this with a weighting function w(a), where a is the age of a measurement. If we apply such a weighted average operation at every moment, we obtain a new function providing s a smoothed estimate of the position of the spaceship:

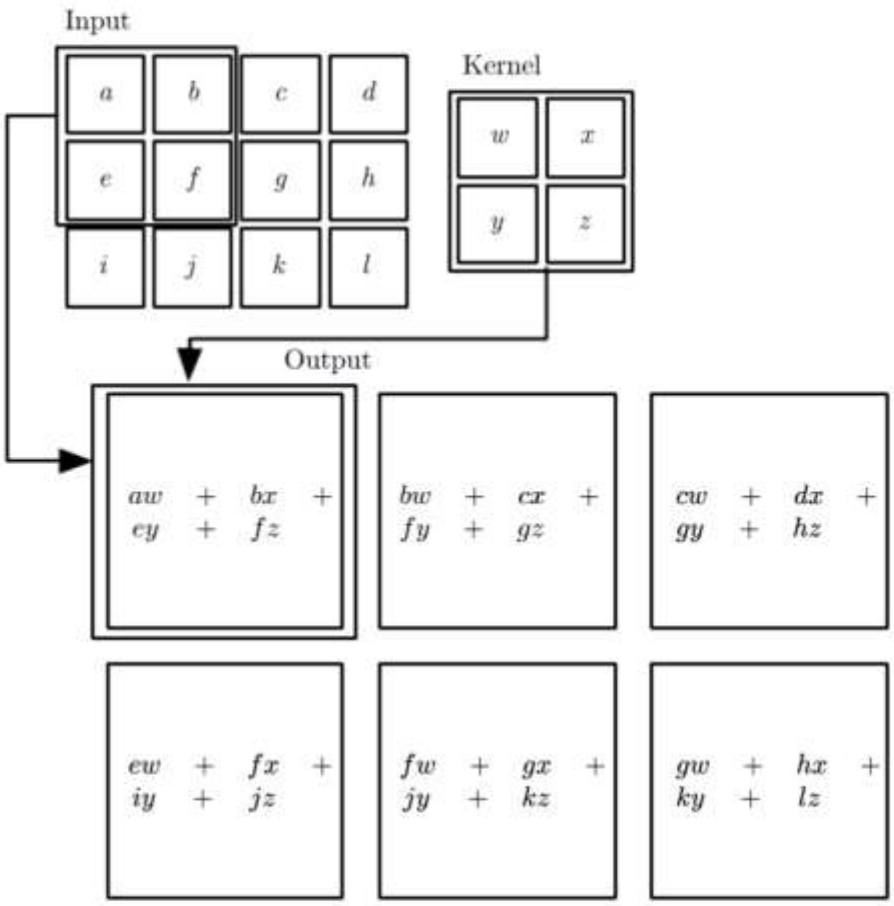
$$s(t) = \int x(a)w(t-a)da$$

In convolutional network terminology, the first argument (in this example, the function x) to the convolution is often referred to as the **input** and the second argument (in this example, the function w) as the **kernel**. The output is sometimes referred to as the **feature map**.
Usually, when we work with data on a computer, time will be discretized, and our sensor will provide data at regular intervals. The time index t can then take on only integer values. If we now assume that x and w are defined only on integer t, we can define the discrete convolution:

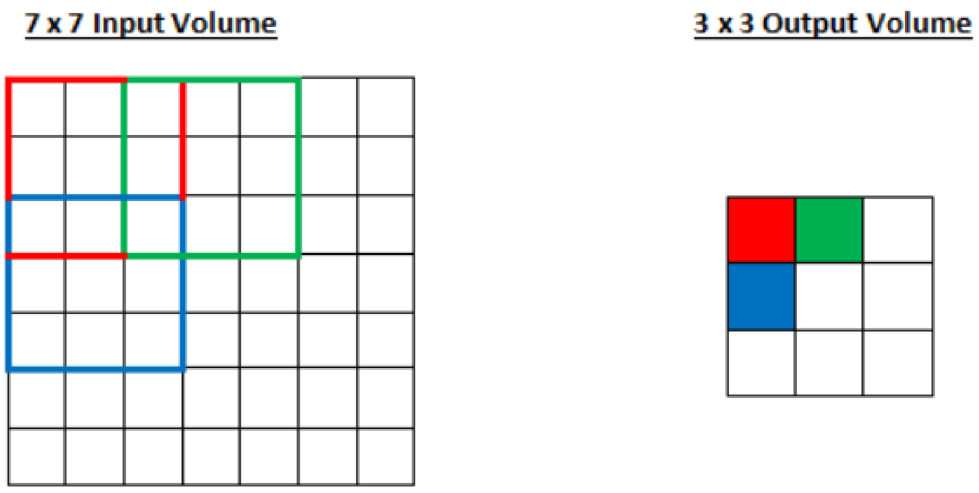
$$s(t) = \sum_{a=-\infty}^{\infty} x(a) * w(t-a)$$

In machine learning applications, the input is usually a multidimensional array of data and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. We will refer to these multidimensional arrays as tensors. Because each element of the input and kernel must be explicitly stored separately, we usually assume that these functions are zero everywhere but the finite set of points for which we store the values. This means that in practice we can implement the infinite summation as a summation over a finite number of array elements.”

Finally, we often use convolutions over more than one axis at a time. For example, if we use a two-dimensional image I as our input, we probably also want to use a two-dimensional kernel K:

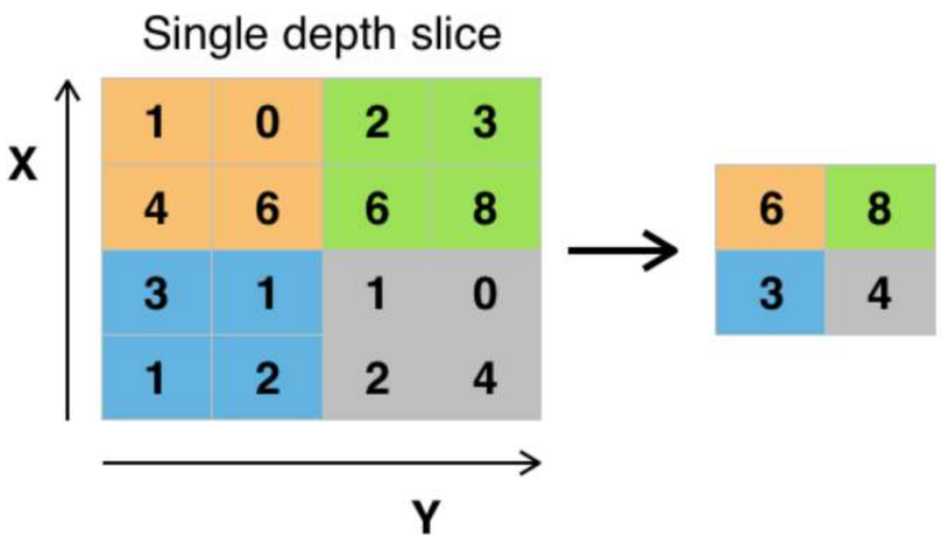


Another example of a 2-dimensional cnn filter is presented bellow, where we have a filter or kernel with dimensions 3x3x1 and stride 2 applied at a 7x7x1 matrix.



2.1.1.1 Pooling

After a CNN layer, it may be beneficial to apply a pooling layer**,** also referred to as a down-sampling layer. There are also several layer options, with max-pooling being the most popular. This basically takes a filter (normally of size 2x2) and a stride of the same length. It then applies it to the input volume and outputs the maximum number in every subregion that the filter convolves around.



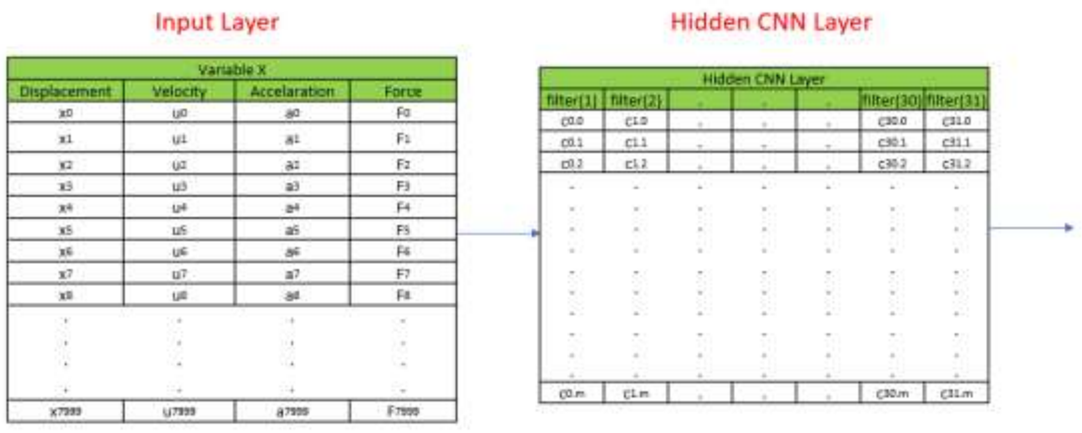
Other options for pooling layers are average pooling and L2-norm pooling. The intuitive reasoning behind this layer is that once we know that a specific feature is in the original input volume (there will be a high activation value), its exact location is not as important as its relative location to the other features. As you can imagine, this layer drastically reduces the spatial dimension (the length and the width change but not the depth) of the input volume. This serves two main purposes. The first is that the amount of parameters or weights is reduced by 75%, thus lessening the computation cost. The second is that it often helps significantly against overfitting.

2.1.2 Example on 1-dimensional Convolution

Using he same example as in chapter 2.1.1. But this time we run a model with a hidden CNN layer with 32 filters and a max pooling layer.

2.1.2.1 Model

- For one or more epochs
    - For each training sample i

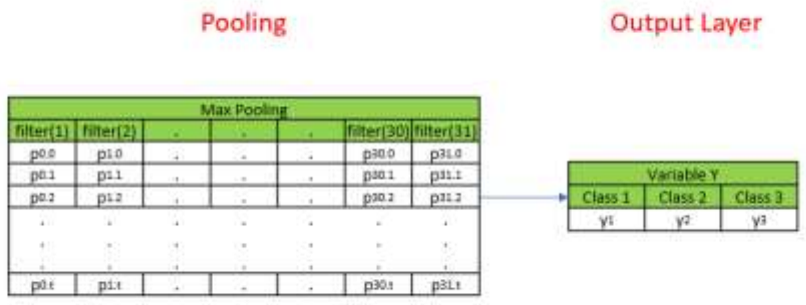


Input Layer

Variable X			
Displacement	Velocity	Accelaration	Force
x0	u0	a0	F0
x1	u1	a1	F1
x2	u2	a2	F2
x3	u3	a3	F3
x4	u4	a4	F4
x5	u5	a5	F5
x6	u6	a6	F6
x7	u7	a7	F7
x8	u8	a8	F8
.	.	.	.
x7999	u7999	a7999	F7999

Hidden CNN Layer

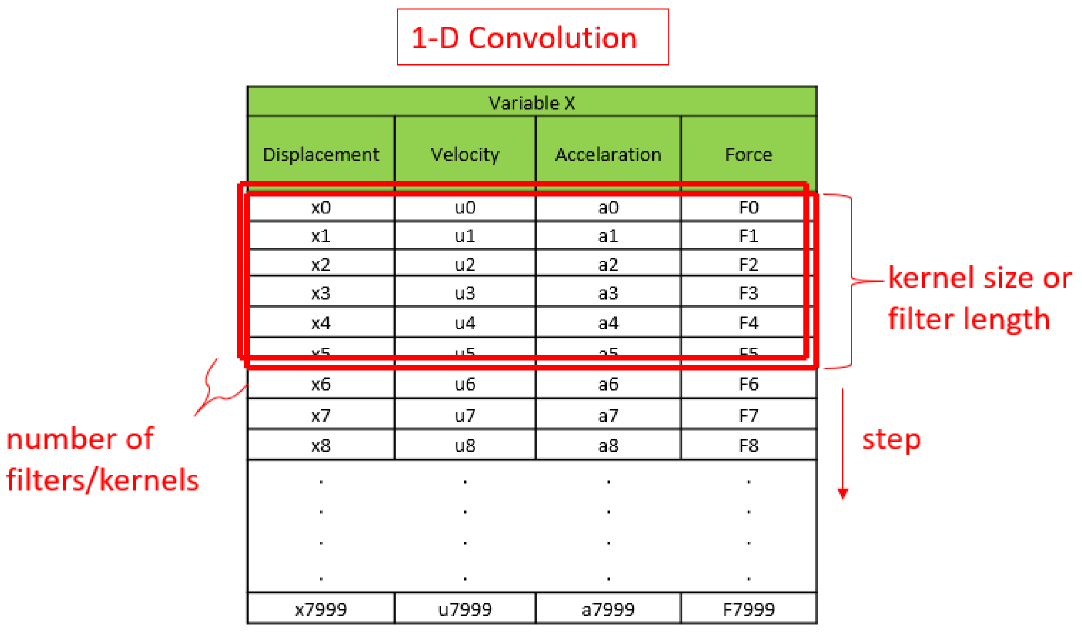
Hidden CNN Layer						
filter[1]	filter[2]	.	.	.	filter[30]	filter[31]
c0.0	c1.0	.	.	.	c30.0	c31.0
c0.1	c1.1	.	.	.	c30.1	c31.1
c0.2	c1.2	.	.	.	c30.2	c31.2
.	.	.	.	.	.	.
c0.m	c1.m	.	.	.	c30.m	c31.m



Max Pooling						
filter(1)	filter(2)	.	.	.	filter(30)	filter(31)
p0.0	p1.0	.	.	.	p30.0	p31.0
p0.1	p1.1	.	.	.	p30.1	p31.1
p0.2	p1.2	.	.	.	p30.2	p31.2
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
p0.t	p1.t	.	.	.	p30.t	p31.t

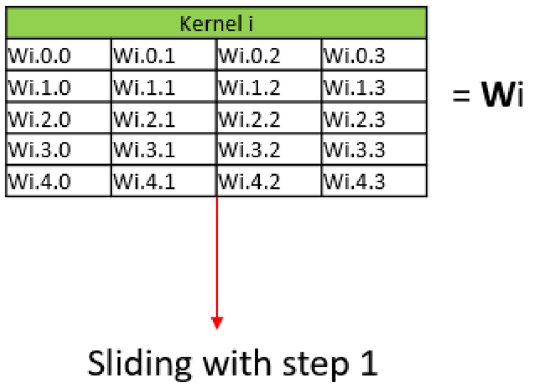
Variable Y		
Class 1	Class 2	Class 3
y1	y2	y3

2.1.2.2 1 Dimensional Convolution



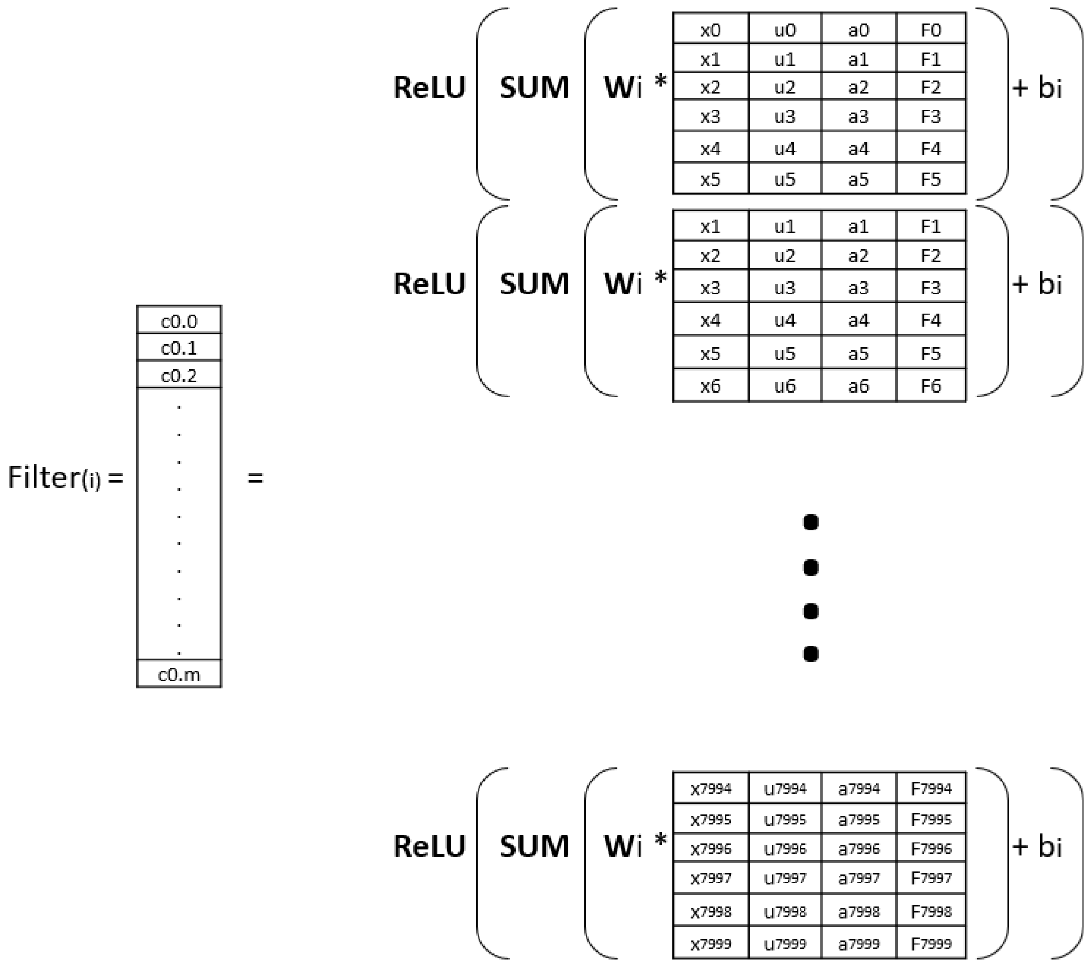
Variable X			
Displacement	Velocity	Accelaration	Force
x0	u0	a0	F0
x1	u1	a1	F1
x2	u2	a2	F2
x3	u3	a3	F3
x4	u4	a4	F4
x5	u5	a5	F5
x6	u6	a6	F6
x7	u7	a7	F7
x8	u8	a8	F8
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
x7999	u7999	a7999	F7999

Assuming we choose a filter/kernel size of 5 and a step of 1. The filter/kernel dimensions will be:



Kernel i			
Wi.0.0	Wi.0.1	Wi.0.2	Wi.0.3
Wi.1.0	Wi.1.1	Wi.1.2	Wi.1.3
Wi.2.0	Wi.2.1	Wi.2.2	Wi.2.3
Wi.3.0	Wi.3.1	Wi.3.2	Wi.3.3
Wi.4.0	Wi.4.1	Wi.4.2	Wi.4.3

To get the values for the activation map we multiply the filter with the corresponding values and summarize the values of the matrix.

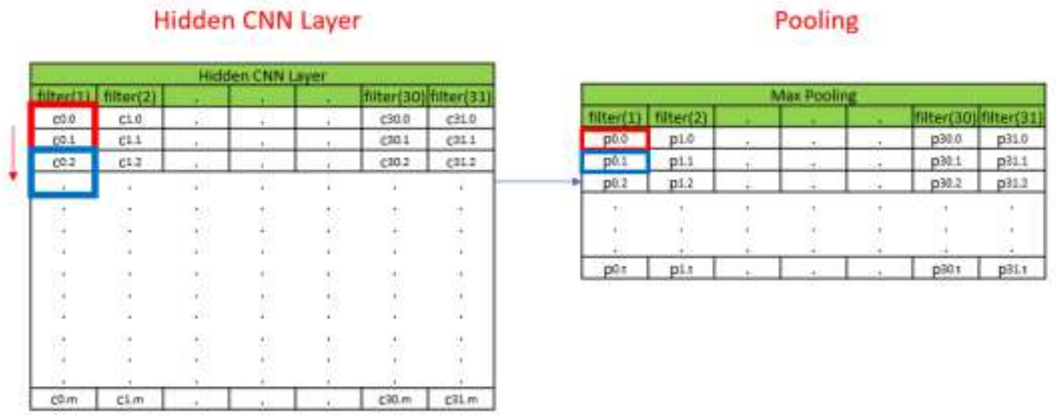


The dimension m of the filter will be the number of times the filter slides across the input, 7996. The parameters that need to be learned here are 32 filters x 20 parameters per filter + 32 biases = 672 weights.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_2 (Conv1D)            (None, 7996, 32)          672
_________________________________________________________________
max_pooling1d_2 (MaxPooling1 (None, 3998, 32)          0
_________________________________________________________________
dense_4 (Dense)              (None, 3998, 3)           99
=================================================================
Total params: 771
Trainable params: 771
Non-trainable params: 0
_________________________________________________________________
```

2.1.2.3 Max Pooling

In this example we will use a 1-dimensional max-pooling layer with dimension 2 and stride 2.



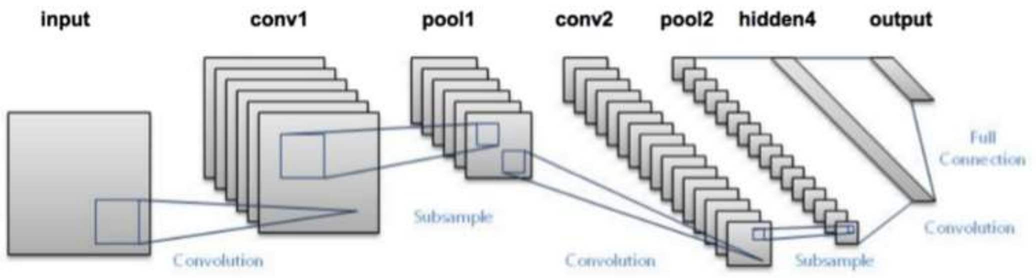
where $p_{0.0} = max\{c_{0.0}, c_{0.1}\}$
$p_{0.1} = max\{c_{0.2}, c_{0.3}\}$
etc.

2.2 CNN Architectures

Choosing the architecture of a CNN algorithm is a complicated and active field of research. It is useful to start from some established algorithms and modifying them through trial and error in order to get a satisfying model. Below some famous CNN architectures are presented from https://medium.com/@sidereal/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5.

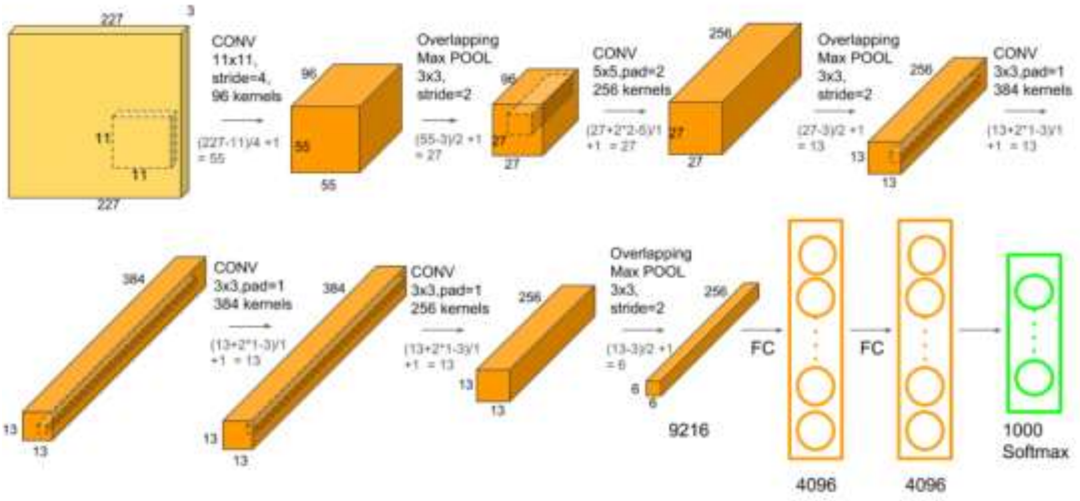
2.2.1 LeNet-5 (1998)

LeNet-5, a pioneering 7-level convolutional network by LeCun et al in 1998, that classifies digits, was applied by several banks to recognize hand-written numbers on checks digitized in 32x32 pixel greyscale input images. The ability to process higher resolution images requires larger and more convolutional layers, so this technique is constrained by the availability of computing resources.



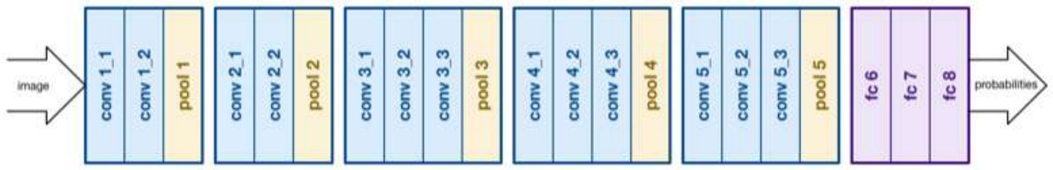
2.2.2 AlexNet (2012)

In 2012, AlexNet significantly outperformed all the prior competitors and won the challenge by reducing the top-5 error from 26% to 15.3%. The second place top-5 error rate, which was not a CNN variation, was around 26.2%.
The network had a very similar architecture as LeNet by Yann LeCun et al but was deeper, with more filters per layer, and with stacked convolutional layers. It consisted 11x11, 5x5,3x3, convolutions, max pooling, dropout, data augmentation, ReLU activations, SGD with momentum. It attached ReLU activations after every convolutional and fully-connected layer. AlexNet was trained for 6 days simultaneously on two Nvidia Geforce GTX 580 GPUs which is the reason for why their network is split into two pipelines. AlexNet was designed by the SuperVision group, consisting of Alex Krizhevsky, Geoffrey Hinton, and Ilya Sutskever.



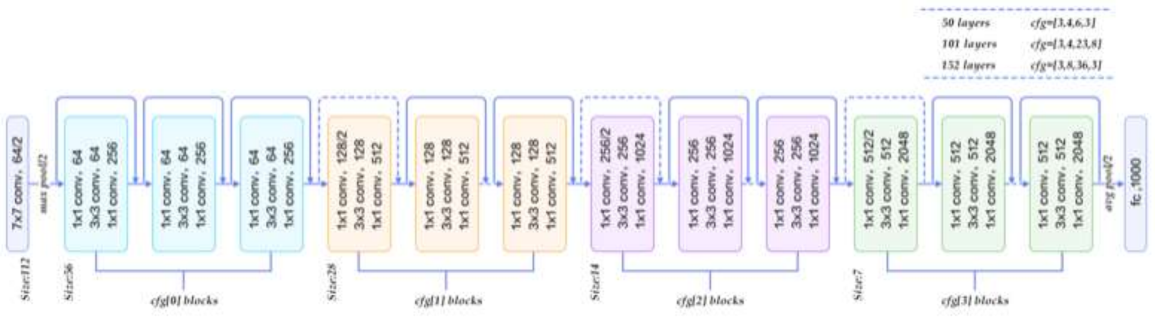
2.2.3 VGGNet(2014)

The runner-up at the ILSVRC 2014 competition is dubbed VGGNet by the community and was developed by Simonyan and Zisserman . VGGNet consists of 16 convolutional layers and is very appealing because of its very uniform architecture. Similar to AlexNet, only 3x3 convolutions, but lots of filters. Trained on 4 GPUs for 2–3 weeks. It is currently the most preferred choice in the community for extracting features from images. The weight configuration of the VGGNet is publicly available and has been used in many other applications and challenges as a baseline feature extractor. However, VGGNet consists of 138 million parameters, which can be a bit challenging to handle.



2.2.4 ResNet

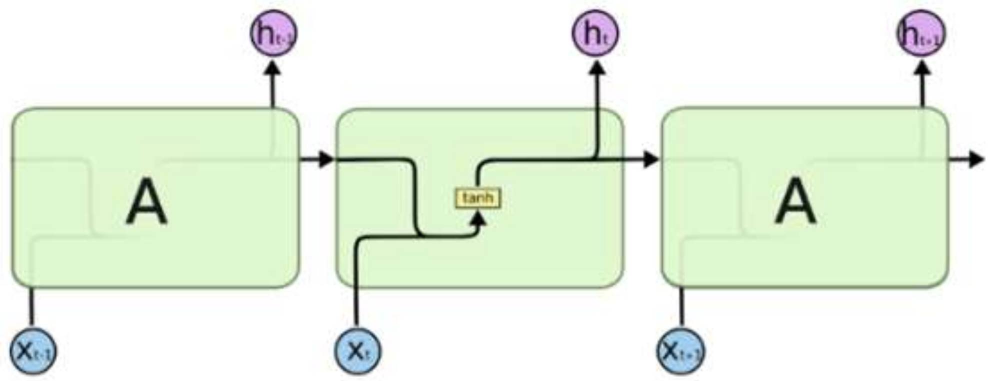
At last, at the ILSVRC 2015, the so-called Residual Neural Network (ResNet) by Kaiming He et al introduced a novel architecture with “skip connections” and features heavy batch normalization. Such skip connections are also known as gated units or gated recurrent units and have a strong similarity to recent successful elements applied in RNNs. Thanks to this technique they were able to train a NN with 152 layers while still having lower complexity than VGGNet. It achieves a top-5 error rate of 3.57% which beats human-level performance on this dataset.



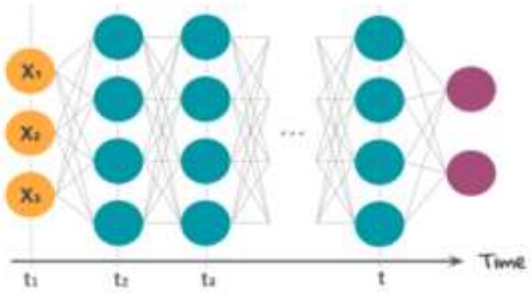
3 Recurrent Neural Networks

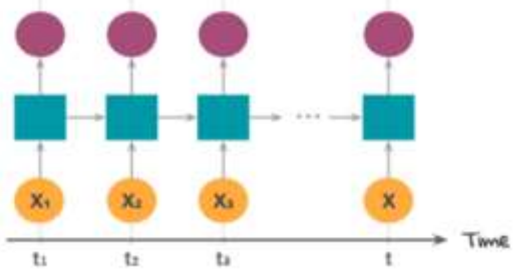
Recurrent neural networks or RNNs (Rumelhart et al., 1986a) are a family of neural networks for processing sequential data. Much as a convolutional network is a neural network that is specialized for processing a grid of values $\boldsymbol{X}$ such as an image, a recurrent neural network is a neural network that is specialized for processing a sequence of values $\boldsymbol{x}(1), \dots \dots, \ \boldsymbol{x}(\tau)$. Just as convolutional networks can readily scale to images with large width and height, and some convolutional networks can process images of variable size, recurrent networks can scale to much longer sequences than would be practical for networks without sequence-based specialization. Most recurrent networks can also process sequences of variable length.

Recurrent means the output at the current time step becomes the input to the next time step. At each element of the sequence, the model considers not just the current input, but what it remembers about the preceding elements. The decision a recurrent net reached at time step $t-1$ affects the decision it will reach one moment later at time step t. So recurrent networks have two sources of input, the present and the recent past, which combine to determine how they respond to new data. Adding memory to neural networks has a purpose: There is information in the sequence itself, and recurrent nets use it to perform tasks that feedforward networks can't. That sequential information is preserved in the recurrent network's hidden state, which manages to span many time steps as it cascades forward to affect the processing of each new example. It is finding correlations between events separated by many moments, and these correlations are called "long-term dependencies", because an event downstream in time depends upon, and is a function of, one or more events that came before. One way to think about RNNs is this: they are a way to share weights over time.

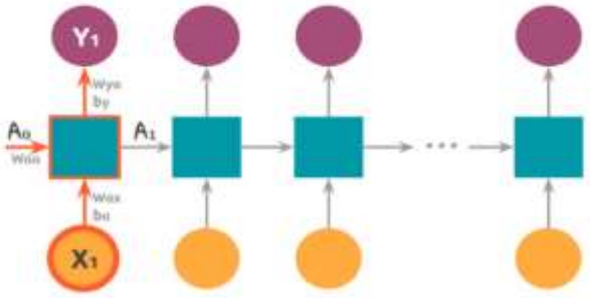


However, it is important to note that sequence models are a bit different from the other neural networks discussed previously. That is because in the case of RNN the data aren't inputted at the same time. The figure bellow describes a time series processed by a feedforward neural network on the left and by a recurrent neural network on the right. As shown, in the RNN the input $\boldsymbol{X}(t=1)$ is introduced to the model first, followed then by input $\boldsymbol{X}(t=2)$, which is influenced by the result of $\boldsymbol{X}(t=1)$. Similarly, $\boldsymbol{X}(t=3)$ is computed with the result from $\boldsymbol{X}(t=2)$ computation stage. Therefore when it comes to data, 'sequential' means we have an order in time between the data. In feed forward neural networks, there isn't any concept of order in $\boldsymbol{x}(t=1)$, $\boldsymbol{x}(t=2)$ and $\boldsymbol{x}(t=3)$. We just input them at once. In the case of RNN, however, they are inputted at different times. Therefore if we change the order, it becomes significantly different.



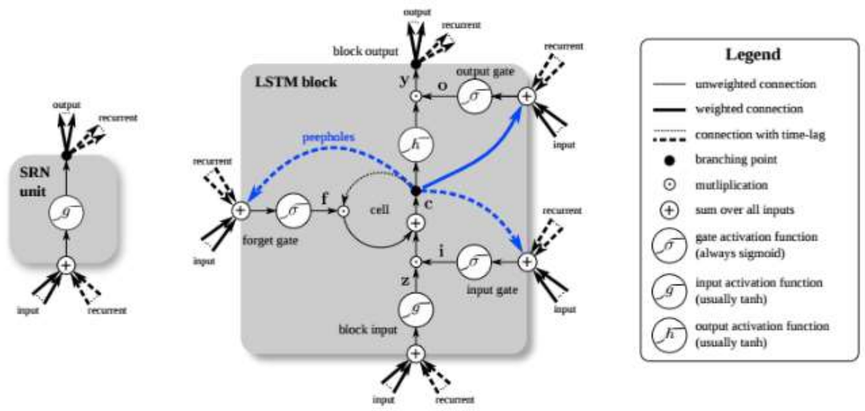


Then for each time step the network receives as input the value $\boldsymbol{X}(t)$ from the input, but also $\boldsymbol{A}(t-1)$ which contains information from previous steps. Naturally, aside from the output of the step $\boldsymbol{Y}(t)$, the network passes on the hidden state $\boldsymbol{A}(t)$.



$$\boldsymbol{A}_t = g(\boldsymbol{W}_{ax} * \boldsymbol{X}_t + \boldsymbol{W}_{aa} * \boldsymbol{A}_{t-1} + \boldsymbol{b}_a)$$
$$\boldsymbol{Y}_t = g(\boldsymbol{W}_{ya} * \boldsymbol{A}_t + \boldsymbol{b}_y)$$

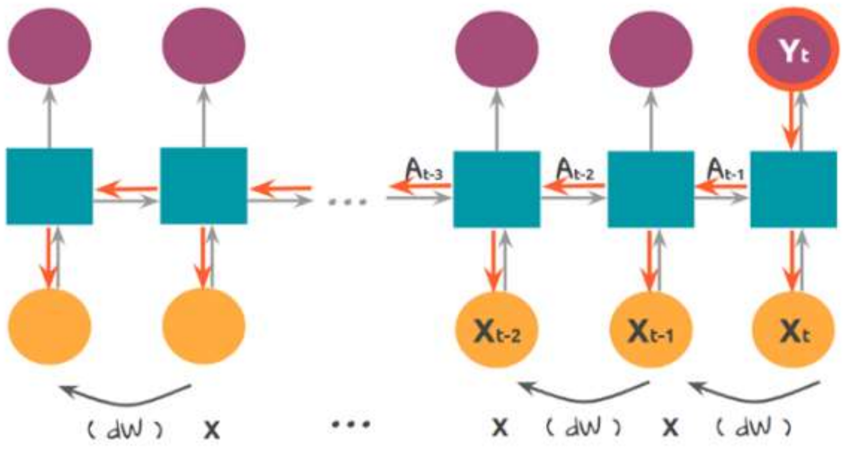
However, the basic RNN approach is often difficult to optimize, which more advanced iterations of the approach aim to fix. As of the writing of this work, the most effective sequence models used in practical applications are called gated RNNs. These include the long short-term memory and networks based on the gated recurrent units. Simple RNNs, as well as GRUs and LSTMs will be presented in later chapters.



3.1.1 Back Propagation Through Time (BPTT)

Backpropagation in feedforward networks moves backward from the final error through the outputs, weights and inputs of each hidden layer, assigning those weights responsibility for a portion of the error by calculating their partial derivatives $-\partial E/\partial W$, or the relationship between their rates of change. Those derivatives are then used by our learning rule, gradient descent, to adjust the weights up or down, whichever direction decreases error.
Recurrent networks rely on an extension of backpropagation called **backpropagation through time**, or **BPTT**. Time, in this case, is simply expressed by a well-defined, ordered

series of calculations linking one time step to the next, which is all backpropagation needs to work.



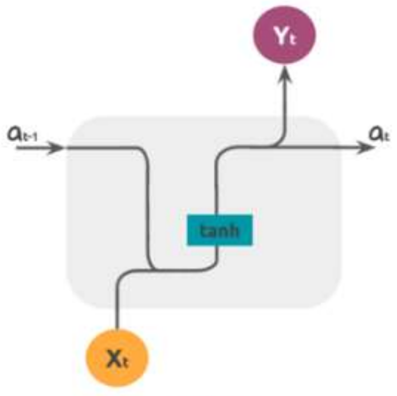
Neural networks, whether they are recurrent or not, are simply nested composite functions like $-f\left(g(h(x))\right)$. Adding a time element only extends the series of functions for which we calculate derivatives with the chain rule.
Conceptually, BPTT works by unrolling all input timesteps. Each timestep has one input timestep, one copy of the network, and one output. Errors are then calculated and accumulated for each timestep. The network is rolled back up and the weights are updated. Spatially, each timestep of the unrolled recurrent neural network may be seen as an additional layer given the order dependence of the problem and the internal state from the previous timestep is taken as an input on the subsequent timestep. BPTT can be computationally expensive as the number of timesteps increases.

Vanishing Gradient Problem

Vanishing gradient problem is a difficulty found in training artificial neural networks with gradient-based learning methods and backpropagation. In such methods, each of the neural network's weights receives an update proportional to the partial derivative of the error function with respect to the current weight in each iteration of training. The problem is that in some cases, the gradient will be vanishingly small, effectively preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training. As one example of the problem cause, traditional activation functions such as the hyperbolic tangent function have gradients in the range (0, 1), and backpropagation computes gradients by the chain rule. This has the effect of multiplying n of these small numbers to compute gradients of the "front" layers in an n-layer network, meaning that the gradient (error signal) decreases exponentially with n while the front layers train very slowly. This problem appears often in BPTT and thus improvements on the RNN concepts have been introduced to deal with this issue.

3.2 Simple RNN Units

The first and , arguably, least useful unit is the simple RNN.



‹ RNN ›

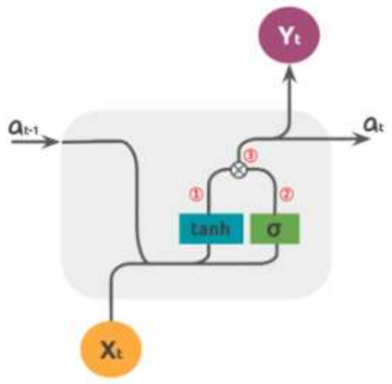
Inputs to the sRNN cell at any step are $\boldsymbol{X}_t$ (current input) and $\boldsymbol{a}_{t-1}$ (previous hidden state).
Outputs from the sRNN cell are $\boldsymbol{a}_t$ (current hidden state) and $\boldsymbol{Y}_t$ (current output).
The unit calculates the hidden state and the current steps output as follows.

$\boldsymbol{a}_t = tanh(\boldsymbol{W}_a * [\boldsymbol{a}_{t-1}, \boldsymbol{X}_t] + \boldsymbol{b}_a)$ **(1)**

$\boldsymbol{Y}_t = g(\boldsymbol{W}_y * \boldsymbol{a}_t + \boldsymbol{b}_y)$ **(2)**

3.3 Gated Recurrent Units

One attempt at countering the vanishing gradient is the Gated Recurrent Unit, or GRU. GRU attempts to remember only the important information and forget the rest. This is done with a sigmoid function or "gate controller" with values between 0 and 1 that decides whether the unit should remember and save at the hidden state or not.



‹ GRUs ›

$\widehat{\boldsymbol{C}}_t = tanh(\boldsymbol{W}_a[\boldsymbol{a}_{t-1}, \boldsymbol{X}_t] + \boldsymbol{b}_a)$ **(1)**

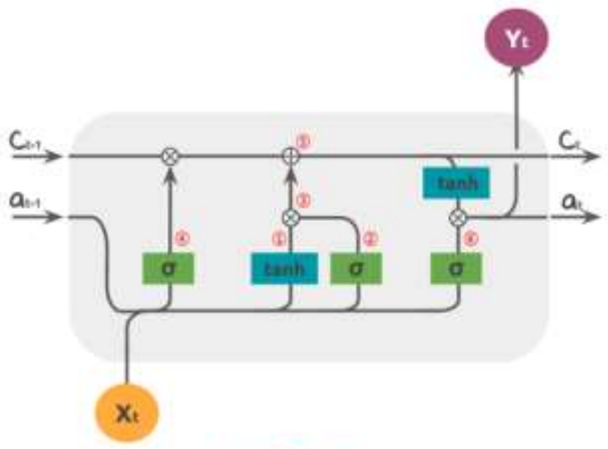
$\boldsymbol{i}_t = \sigma(\boldsymbol{W}_i[\boldsymbol{a}_{t-1}, \boldsymbol{X}_t] + \boldsymbol{b}_\Gamma)$ **(2)**

$\boldsymbol{a}_t = \boldsymbol{i}_t * \widehat{\boldsymbol{C}}_t + (1 - \boldsymbol{i}_t) * \boldsymbol{a}_{t-1}$ **(3)**

$\boldsymbol{Y}_t = g(\boldsymbol{W}_y * \boldsymbol{a}_t + \boldsymbol{b}_y)$ **(4)**

3.4 Long-Short-Term-Memory Units

In the mid-90s, a variation of recurrent net with so-called Long Short-Term Memory units, or LSTMs, was proposed by the German researchers Sepp Hochreiter and Juergen Schmidhuber as a solution to the vanishing gradient problem. LSTMs help preserve the error that can be backpropagated through time and layers. By maintaining a more constant error, they allow recurrent nets to continue to learn over many time steps (over 1000), thereby opening a channel to link causes and effects remotely. This is one of the central challenges to machine learning and AI, since algorithms are frequently confronted by environments where reward signals are sparse and delayed, such as life itself.
LSTMs contain information outside the normal flow of the recurrent network in a gated cell. Information can be stored in, written to, or read from a cell. The cell makes decisions about what to store, and when to allow reads, writes and erasures, via gates that open and close. Those gates act on the signals they receive, and similar to the neural network's nodes, they block or pass on information based on its strength and import, which they filter with their own sets of weights. Those weights, like the weights that modulate input and hidden states, are adjusted via the recurrent networks learning process. That is, the cells learn when to allow data to enter, leave or be deleted through the iterative process of making guesses, backpropagating error, and adjusting weights via gradient descent.
A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell is responsible for "remembering" values over arbitrary time intervals; hence the word "memory" in LSTM. Each of the three gates can be thought of as a "conventional" artificial neuron, as in a multi-layer (or feedforward) neural network: that is, they compute an activation (using an activation function) of a weighted sum. There are connections between these gates and the cell. The expression long short-term refers to the fact that LSTM is a model for the short-term memory which can last for a long period of time. An LSTM is well-suited to classify, process and predict time series given time lags of unknown size and duration between important events.



‹ LSTM ›

Inputs to the LSTM cell at any step are $\boldsymbol{X}_t$ (current input) , $\boldsymbol{a}_{t-1}$ (previous hidden state) and $\boldsymbol{C}_{t-1}$ (previous memory state). Outputs from the LSTM cell are $\boldsymbol{a}_t$ (current hidden state), $\boldsymbol{C}_t$ (current memory state) and $\boldsymbol{Y}_t$ (current output).

The input gate $\boldsymbol{i}_t$ takes the previous output and the new input and passes them through a sigmoid layer. This gate returns a value between 0 and 1. $\widehat{\boldsymbol{C}}_t$ is called the candidate layer and performs a tanh transformation, which returns a value from -1 to 1.

$\widehat{\boldsymbol{C}}_t = tanh(\boldsymbol{W}_a[\boldsymbol{a}_{t-1}, \boldsymbol{X}_t] + \boldsymbol{b}_a)$ **(1)**

$\boldsymbol{i}_t = \sigma(\boldsymbol{W}_i[\boldsymbol{a}_{t-1}, \boldsymbol{X}_t] + \boldsymbol{b}_i)$ **(2)**

The forget gate $\boldsymbol{f}_t$ is another sigmoid layer that takes the output at $t-1$ and the current input at time t and concatenates them into a single tensor and applies a linear transformation followed by a sigmoid. This number is multiplied with the internal state and that is why the gate is called a forget gate. If $f_t = 0$ then the previous internal state is completely forgotten, while if $f_t = 1$ it will be passed through unaltered.

$\boldsymbol{f}_t = \sigma(\boldsymbol{W}_f[\boldsymbol{a}_{t-1}, \boldsymbol{X}_t] + \boldsymbol{b}_f)$ **(4)**

$\boldsymbol{C}_t = \boldsymbol{f}_t * \boldsymbol{C}_{t-1} + \boldsymbol{i}_t * \widehat{\boldsymbol{C}}_t$ **(3)**

The output gate $\boldsymbol{o}_t$ controls how much of the internal state is passed to the output and it works in a similar way to the other gates.

$\boldsymbol{o}_t = \sigma(\boldsymbol{W}_o[\boldsymbol{a}_{t-1}, \boldsymbol{X}_t] + \boldsymbol{b}_o)$ **(5)**

Then the hidden state, that is passed to the next time step, and the output of the current time step are calculated as follows.

$\boldsymbol{a}_t = \boldsymbol{a}_t * \tanh(\boldsymbol{C}_t)$ **(6)**

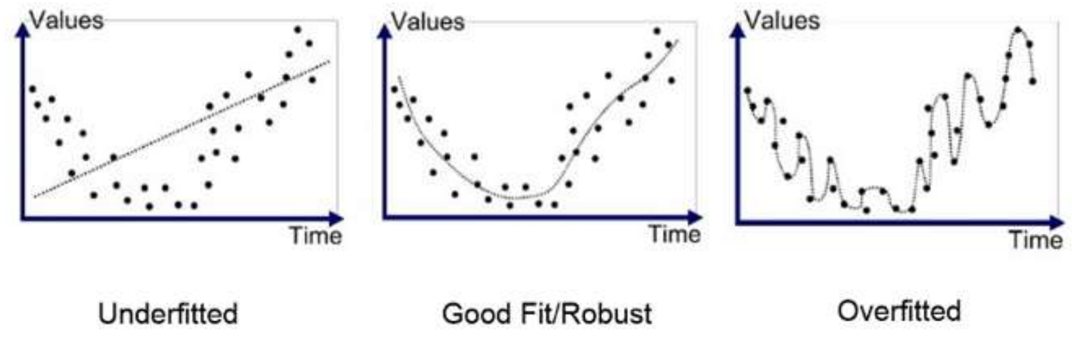
$\boldsymbol{Y}_t = g(\boldsymbol{W}_y * \boldsymbol{a}_t + \boldsymbol{b}_y)$ **(7)**

4 Evaluation and Regularization

The central challenge in machine learning is that we must perform well on new, previously unseen inputs—not just those on which our model was trained. The ability to perform well on previously unobserved inputs is called **generalization**.
Of course, when we use a machine learning algorithm, we do not fix the parameters ahead of time, then sample both datasets. We sample the training set, then use it to choose the parameters to reduce training set error, then sample the test set. Under this process, the expected test error is greater than or equal to the expected value of training error. The factors determining how well a machine learning algorithm will perform are its ability to:

- Make the training error small.
- Make the gap between training and test error small.

These two factors correspond to the two central challenges in machine learning, **underfitting** and **overfitting** .



Overfitting happens when a model learns the detail and noise in the training data to the extent that it negatively impacts the performance of the model on new data. This means that the noise or random fluctuations in the training data is picked up and learned as concepts by the model. The problem is that these concepts do not apply to new data and negatively impact the models ability to generalize.

Underfitting refers to a model that can neither model the training data nor generalize to new data. An underfit machine learning model is not a suitable model and will be obvious as it will have poor performance on the training data. Underfitting is often not discussed as it is easy to detect given a good performance metric. The remedy is to move on and try alternate machine learning algorithms. Nevertheless, it does provide a good contrast to the problem of overfitting.

4.1 Evaluating Performance

4.1.1 Validation Set

As was already explained, we are interested in how well the machine learning algorithm performs on data that it has not seen before, since this determines how well it will work when deployed in the real world. We therefore evaluate these performance measures using a **test set** of data that is separate from the data used for training the machine learning system. As a test set we can utilize:

- Training Dataset

Prepare your model on the entire training dataset, then evaluate the model on the same dataset. This is generally problematic not least because a perfect algorithm could game this evaluation technique by simply memorizing (storing) all training patterns and achieve a perfect score, which would be misleading.

- Supplied Test Set

Split your dataset manually using another program. Prepare your model on the entire training dataset and use the separate test set to evaluate the performance of the model. This is a good approach if you have a large dataset (many tens of thousands of instances).

- Percentage Split

Randomly split your dataset into a training and a testing partitions each time you evaluate a model. This can give you a very quick estimate of performance and like using a supplied test set, is preferable only when you have a large dataset.

- Cross Validation

Split the dataset into k-partitions or folds. Train a model on all of the partitions except one that is held out as the test set, then repeat this process creating k-different models and give each fold a chance of being held out as the test set. Then calculate the average performance of all k models. This is the gold standard for evaluating model performance, but has the cost of creating many more models.

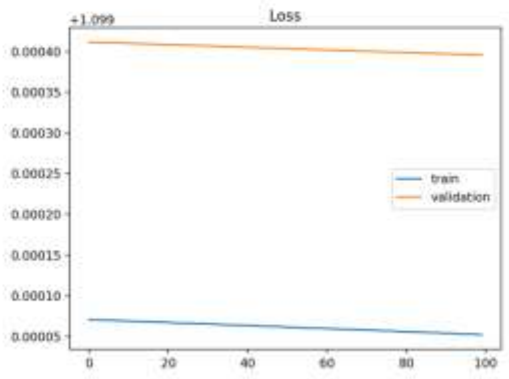
4.1.2 Learning curves

Learning curves are a line plot of learning (*y*-axis) over experience (*x*-axis). In practice, learning is the result of the **loss** function and experience is the epochs (passing over dataset). It is usually recorded on the training dataset, to give an idea of how well the model is "learning", and on the validation dataset, to get an idea of how well the model is "generalizing."

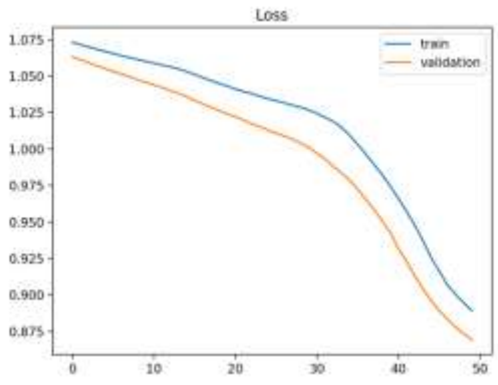
Learning curves are a widely used diagnostic tool in machine learning for algorithms that learn from a training dataset incrementally and are critical in understanding the shortcomings of a model. A gentle introduction to learning curves for diagnosing machine learning model performance can be found at https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/

- Underfit Learning Curves

An underfit model can be identified from the learning curve of the training loss only.
It may show a flat line or noisy values of relatively high loss, indicating that the model was unable to learn the training dataset at all.



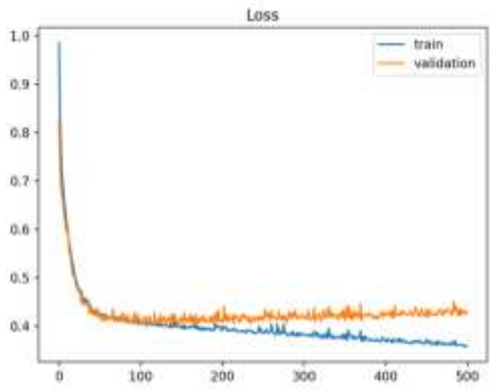
An underfit model may also be identified by a training loss that is decreasing and continues to decrease at the end of the plot. This indicates that the model is capable of further learning and possible further improvements and that the training process was halted prematurely



- Overfit Learning Curves

Overfitting refers to a model that has learned the training dataset too well, including the statistical noise or random fluctuations in the training dataset. A plot of learning curves shows overfitting if:

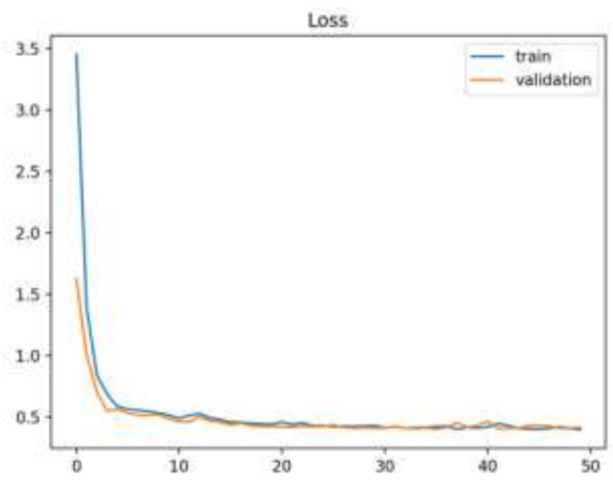
- The plot of training loss continues to decrease with experience.
- The plot of validation loss decreases to a point and begins increasing again.



- Good Fit Learning Curves

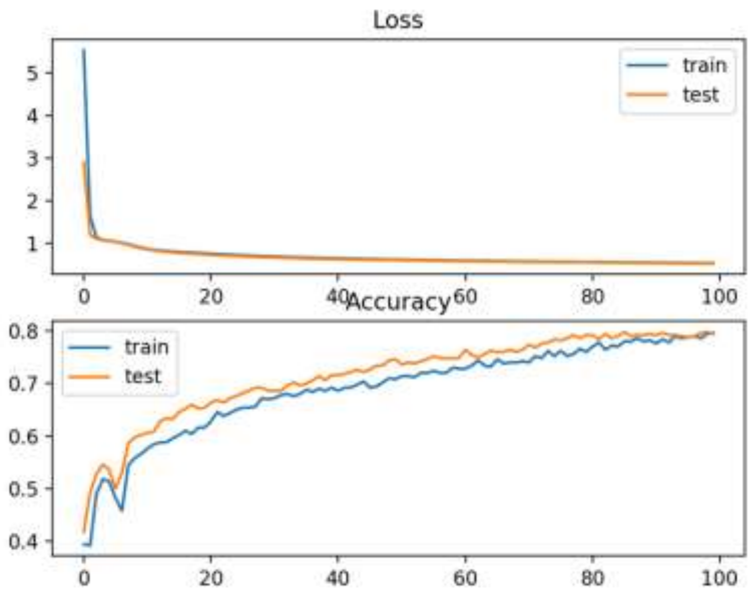
A good fit is the goal of the learning algorithm and exists between an overfit and underfit model. A good fit is identified by a training and validation loss that decreases to a point of stability with a minimal gap between the two final loss values. The loss of the model will almost always be lower on the training dataset than the validation dataset. This means that we should expect some gap between the train and validation loss learning curves. This gap is referred to as the “generalization gap.” A plot of learning curves shows a good fit if:

- The plot of training loss decreases to a point of stability.
- The plot of validation loss decreases to a point of stability and has a small gap with the training loss.



Accuracy curves
In some cases, it is also common to create learning curves for multiple metrics, such as in the case of **classification** predictive modeling problems, where the model may be optimized according to cross-entropy loss and model performance is evaluated using classification accuracy. In this case, two plots are created, one for the learning curves of each metric, and each plot can show two learning curves, one for each of the train and validation datasets. Accuracy is just the proportion of examples for which the model produces the correct output.

$$accuracy = \frac{\#\,correct\ predictions}{\#samples}$$

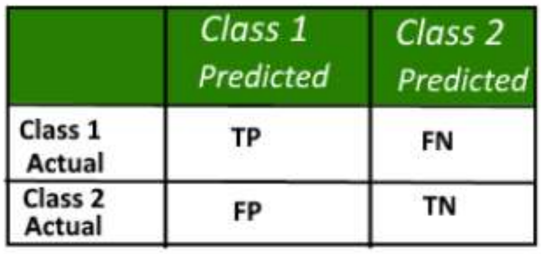


4.1.3 Confusion Matrix

A performance measure particularly useful in **classification** tasks is the confusion matrix. Classification accuracy alone can be misleading if you have an unequal number of observations in each class or if you have more than two classes in your dataset. The main problem with classification accuracy is that it hides the detail you need to better understand the performance of your classification model. There are two examples where you are most likely to encounter this problem:

- When your data has more than 2 classes. With 3 or more classes you may get a classification accuracy of 80%, but you don't know if that is because all classes are being predicted equally well or whether one or two classes are being neglected by the model.
- When your data does not have an even number of classes. You may achieve accuracy of 90% or more, but this is not a good score if 90 records for every 100 belong to one class and you can achieve this score by always predicting the most common class value.

A confusion matrix is a summary of prediction results on a classification problem.
The number of correct and incorrect predictions are summarized with count values and broken down by each class. This is the key to the confusion matrix. The confusion matrix shows the ways in which your classification model is confused when it makes predictions. It gives us insight not only into the errors being made by a classifier but more importantly the types of errors that are being made.



	Class 1 Predicted	Class 2 Predicted
Class 1 Actual	TP	FN
Class 2 Actual	FP	TN

Where

- Positive (*P*) : Observation is positive-correct
- Negative (*N*) : Observation is not positive-false
- True Positive (*TP*) : Observation is positive, and is predicted to be positive.
- False Negative (*FN*) : Observation is positive, but is predicted negative.
- True Negative (*TN*) : Observation is negative, and is predicted to be negative.
- False Positive (*FP*) : Observation is negative, but is predicted positive.

It is easy to deduce that the classification rate or accuracy is given by the relation:

$$\textbf{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

4.2 Regularization Techniques

Many strategies used in machine learning are explicitly designed to reduce the test error, possibly at the expense of increased training error. We can say that any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error is **regularization**.
Some strategies put extra constraints on a machine learning model, such as adding restrictions on the parameter values. Some add extra terms in the objective function that can be thought of as corresponding to a soft constraint on the parameter values.
Developing more effective regularization strategies has been one of the major research efforts in the machine learning field. Sometimes these constraints and penalties are designed to encode specific kinds of prior knowledge. Other times, these constraints and penalties are designed to express a generic preference for a simpler model class in order to

promote generalization. Sometimes penalties and constraints are necessary to make an underdetermined problem determined.

4.2.1 Parameter norm Penalties

Many regularization approaches are based on limiting the capacity of models, such as neural networks, linear regression, or logistic regression, by adding a parameter norm penalty $\Omega(\boldsymbol{\theta})$ to the objective function J. We denote the regularized objective function by $\tilde{J}$:

$$\tilde{J}(\boldsymbol{\theta}, \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}, \boldsymbol{X}, \boldsymbol{y}) + a\,\Omega(\boldsymbol{\theta})$$

where $\alpha \in [0, \infty)$ is a hyperparameter that weights the relative contribution of the norm penalty term, Ω, relative to the standard objective function J. Setting α to 0 results in no regularization. Larger values of α correspond to more regularization. When our training algorithm minimizes the regularized objective function $\tilde{J}$ it will decrease both the original objective J on the training data and some measure of the size of the parameters $\boldsymbol{\theta}$ (or some subset of the parameters). Different choices for the parameter norm Ω can result in different solutions being preferred.

4.2.1.1 L^2 Parameter Regularization

The L^2 parameter norm penalty is commonly known as weight decay. This regularization strategy drives the weights closer to the origin by adding a regularization term $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2$ to the objective function. In other academic communities, L^2 regularization is also known as ridge regression or Tikhonov regularization.

We can gain some insight into the behavior of weight decay regularization by studying the gradient of the regularized objective function. To simplify the presentation, we assume no bias parameter, so $\boldsymbol{\theta}$ is just $\boldsymbol{w}$. Such a model has the following total objective function:

$$\tilde{J}(\boldsymbol{\theta}, \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}, \boldsymbol{X}, \boldsymbol{y}) + \frac{a}{2}\,\boldsymbol{w}^T\boldsymbol{w}$$

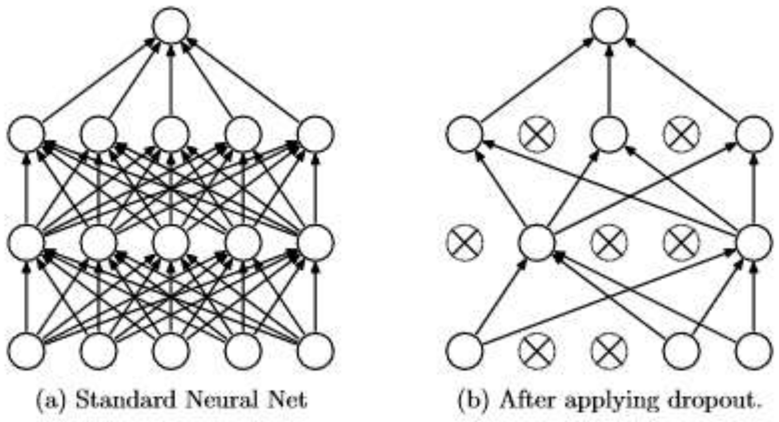
4.2.1.2 L^1 Parameter Regularization

While L^2 weight decay is the most common form of weight decay, there are other ways to penalize the size of the model parameters. Another option is to use L^1 regularization. Formally, L^1 regularization on the model parameter w is defined as $\Omega(\boldsymbol{\theta}) = \|\boldsymbol{w}\|_1 = \sum_i |\boldsymbol{w}_i|$. The objective loss function becomes:

$$\tilde{J}(\boldsymbol{\theta}, \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}, \boldsymbol{X}, \boldsymbol{y}) + a \sum_{\boldsymbol{i}} |\boldsymbol{w}_i|$$

4.2.2 Dropout

Dropout is a technique that prevents overfitting and provides a way of approximately combining exponentially many different neural network architectures efficiently. The term “dropout” refers to dropping out units (hidden and visible) in a neural network. By dropping a unit out, we mean temporarily removing it from the network, along with all its incoming and outgoing connections, as shown in the figure bellow. The choice of which units to drop is random. In the simplest case, each unit is retained with a fixed probability p independent of other units, where p can be chosen using a validation set or can simply be set at 0.5, which seems to be close to optimal for a wide range of networks and tasks. For the input units, however, the optimal probability of retention is usually closer to 1 than to 0.5



Applying dropout to a neural network amounts to sampling a “thinned” network from it. The thinned network consists of all the units that survived dropout (Figure b). A neural net with n units, can be seen as a collection of $2 * n$ possible thinned neural networks. These networks all share weights. For each presentation of each training case, a new thinned network is sampled and trained. So training a neural network with dropout can be seen as training a collection of $2 * n$ thinned networks with extensive weight sharing, where each thinned network gets trained very rarely, if at all.

5 Data preparation

In this work, the focus will be on performing classification and regression tasks, using time series as input. A **time series** is a series of data points indexed (or listed or graphed) in time order. Most commonly, a time series is a sequence taken at successive equally spaced points in time. Thus it is a sequence of discrete-time data. Examples of time series are heights of ocean tides, counts of sunspots, and the daily closing value of the Dow Jones Industrial Average. Time series are very frequently plotted via line charts. Time series are used in statistics, signal processing, pattern recognition, econometrics, mathematical finance, weather forecasting, earthquake prediction, electroencephalography, control engineering, astronomy, communications engineering, and largely in any domain of applied science and engineering which involves temporal measurements.

- **Univariate time-series:** A univariate time series, as the name suggests, is a series with a single time-dependent variable
- **Multivariate time-series**: A Multivariate time series has more than one time-dependent variable. Each variable depends not only on its past values but also has some dependency on other variables.

5.1.1 Input-Output

In this thesis, the layers that will be used are 1-dimensional CNN layers, 2 dimensional CNN layers, LSTM layers and fully connected dense layers. The input to the model will need to match the input to the first layer and the same for the output and the last layer. The dimensions for each layer type are presented bellow. These are based on the documentation for the machine learning library “Keras”.

1-Dimensional CNN layer:

- input 3D tensor-(samples, timesteps, channels)
- output 3D tensor-(samples, new timesteps, channels)

2-Dimensional CNN layer:

- input 4D tensor-(samples, channels, rows, columns)
- output 4D tensor-(samples, filters, new rows, new columns)

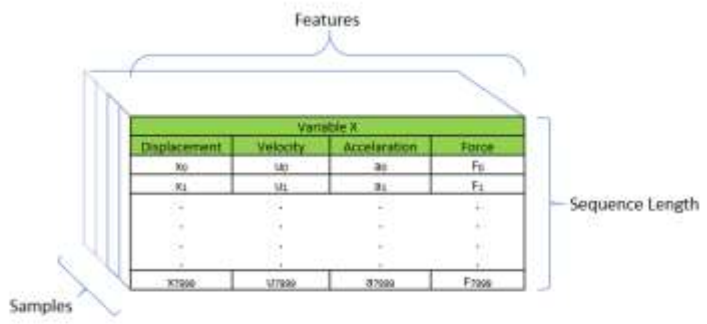
LSTM layer:

- input 3D tensor-(samples, timesteps, input dim).
- output 2D tensor-(samples, units) or 3D- (samples, timesteps, units) if we choose to return the entire sequence

Fully Connected dense layers:

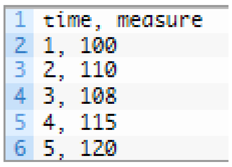
- input nD tensor-(samples , ..., input dim)
- output nD tensor- (samples, ..., units)

Example of a 3D input matrix

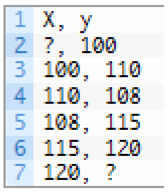


5.1.2 Time-Series Forecasting to Supervised learning

Before neural networks can be used, time series forecasting problems must be re-framed as supervised learning problems. From a sequence to pairs of input and output sequences. There are a number of ways to model forecasting as a supervised learning problem. However, in this work the sliding window method or lag method was used.
A time series is a sequence of numbers that are ordered by a time index. This can be thought of as a list or column of ordered values.


```
time, measure
1, 100
2, 110
3, 108
4, 115
5, 120
```

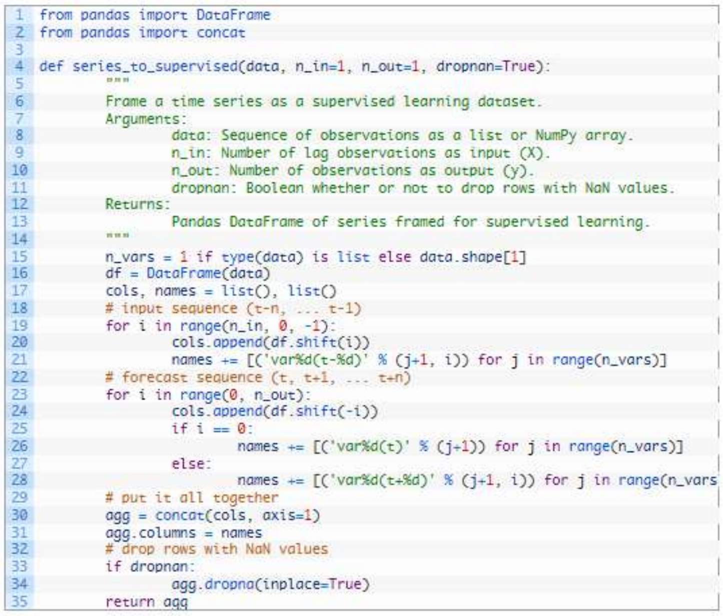
A supervised learning problem is comprised of input patterns $(\boldsymbol{X})$ and output patterns $(\boldsymbol{y})$,such that an algorithm can learn how to predict the output patterns from the input patterns. Given a sequence of numbers for a time series dataset, we can restructure the data to look like a supervised learning problem. We can do this by using previous time steps as input variables and use the next time step as the output variable.


```
X, y
?, 100
100, 110
110, 108
108, 115
115, 120
120, ?
```

The use of prior time steps to predict the next time step is called the **sliding window method**. The number of previous time steps is called the window width or size of the lag. This sliding window is the basis for how we can turn any time series dataset into a supervised learning problem. From this simple example, we can notice a few things:

- The method can work to turn a time series into either a regression or a classification supervised learning problem for real-valued or labeled time series values.
- Once a time series dataset is prepared this way, any of the standard linear and nonlinear machine learning algorithms may be applied, as long as the order of the rows is preserved.
- The width of the sliding window can be increased to include more previous time steps.
- This approach can be used on a time series that has more than one value, or so-called multivariate time series.

Time series were converted to supervised tasks using the following “Python” function from https://machinelearningmastery.com

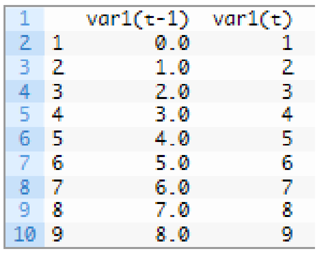


```python
from pandas import DataFrame
from pandas import concat

def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
        """
        Frame a time series as a supervised learning dataset.
        Arguments:
                data: Sequence of observations as a list or NumPy array.
                n_in: Number of lag observations as input (X).
                n_out: Number of observations as output (y).
                dropnan: Boolean whether or not to drop rows with NaN values.
        Returns:
                Pandas DataFrame of series framed for supervised learning.
        """
        n_vars = 1 if type(data) is list else data.shape[1]
        df = DataFrame(data)
        cols, names = list(), list()
        # input sequence (t-n, ... t-1)
        for i in range(n_in, 0, -1):
                cols.append(df.shift(i))
                names += [('var%d(t-%d)' % (j+1, i)) for j in range(n_vars)]
        # forecast sequence (t, t+1, ... t+n)
        for i in range(0, n_out):
                cols.append(df.shift(-i))
                if i == 0:
                        names += [('var%d(t)' % (j+1)) for j in range(n_vars)]
                else:
                        names += [('var%d(t+%d)' % (j+1, i)) for j in range(n_vars
        # put it all together
        agg = concat(cols, axis=1)
        agg.columns = names
        # drop rows with NaN values
        if dropnan:
                agg.dropna(inplace=True)
        return agg
```

The number of time steps ahead to be forecasted is important. Again, it is traditional to use different names for the problem depending on the number of time-steps to forecast:

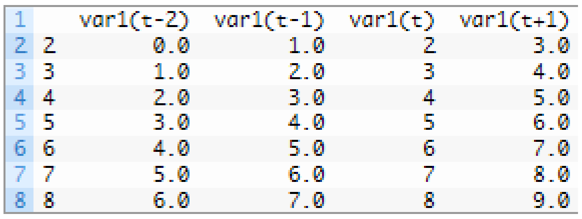
- **One-Step Forecast:** This is where the next time step is predicted.
- **Multi-Step Forecast:** This is where two or more future time steps are to be predicted.

One-Step Univariate Forecasting



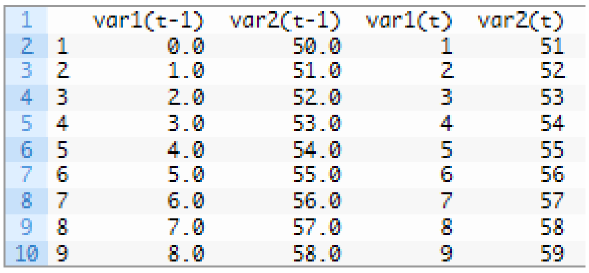
```
   var1(t-1)  var1(t)
1        0.0        1
2        1.0        2
3        2.0        3
4        3.0        4
5        4.0        5
6        5.0        6
7        6.0        7
8        7.0        8
9        8.0        9
```

Multi-Step or Sequence Forecasting



```
  var1(t-2)  var1(t-1)  var1(t)  var1(t+1)
2       0.0        1.0        2        3.0
3       1.0        2.0        3        4.0
4       2.0        3.0        4        5.0
5       3.0        4.0        5        6.0
6       4.0        5.0        6        7.0
7       5.0        6.0        7        8.0
8       6.0        7.0        8        9.0
```

Multivariate and Multi-Step Forecasting



```
   var1(t-1)  var2(t-1)  var1(t)  var2(t)
1        0.0       50.0        1       51
2        1.0       51.0        2       52
3        2.0       52.0        3       53
4        3.0       53.0        4       54
5        4.0       54.0        5       55
6        5.0       55.0        6       56
7        6.0       56.0        7       57
8        7.0       57.0        8       58
9        8.0       58.0        9       59
```

5.1.3 Recurrence plot

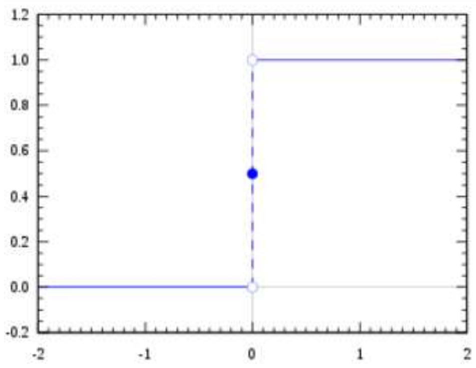
An interesting approach to time-series classification, that was experimented with, is presented in “Classification of Time-Series Images Using Deep Convolutional Neural Networks” by Nima Hatami, Yann Gavet and Johan Debayle. This method uses recurrence plots to represent the time histories in 2d matrices and then uses traditional convolutional neural networks for the classification task.

Recurrence plot – A recurrence plot (RP) is an advanced technique of nonlinear data analysis. It is a visualization (or a graph) of a square matrix, in which the matrix elements correspond to those times at which a state of a dynamical system recurs (columns and rows correspond then to a certain pair of times). Technically, the RP reveals all the times when the phase space trajectory of the dynamical system visits roughly the same area in the phase space.

The RP is a visualization tool that aims to explore the m dimensional phase space trajectory through a 2D representation of its recurrences. The main idea is to reveal in which points some trajectories return to a previous state and it can be formulated as:

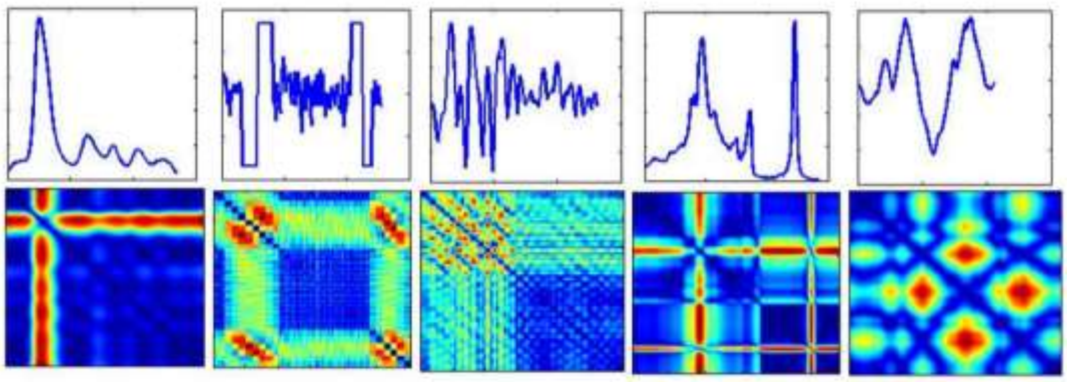
$$R_{i,j} = \theta\left(\varepsilon - \left\|\vec{s}^{i} - \vec{s}^{j}\right\|\right), \qquad \vec{s}(.) \in R^{m}, \quad i,j = 1, \dots \dots \dots, K$$

where K is the number of considered states, $\vec{s}$ is a threshold distance, || . || a norm and θ(.) the Heaviside function. The Heaviside function, or unit step function is a discontinuous function whose value is zero for negative arguments and one for positive arguments.



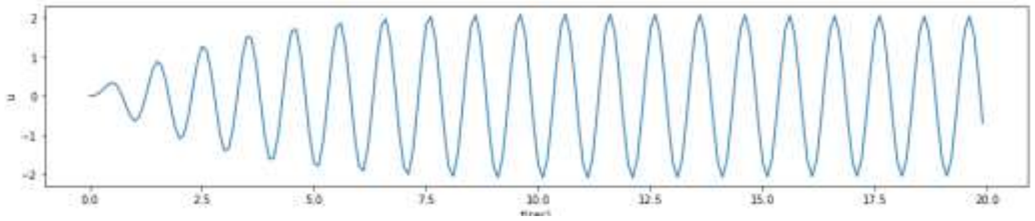
Heaviside function

The *R*-matrix contains both texture which are single dots, diagonal lines as well as vertical and horizontal lines; and typology information which are characterized as homogeneous, periodic, drift and disrupted. For instance, fading to the upper left and lower right corners means that the process contains a trend or drift; or vertical and horizontal lines/clusters show that some states do not change or change slowly for some time and this can be interpreted as laminar states. Obviously, there are patterns and information in RP that are not always very easy to visually see and interpret.



Example

As an example, we will use a univariate time-series.



The duration of the history is 20 seconds with a time step of 0.1. This leads to 200 values.

Step 1 Recurrence plot

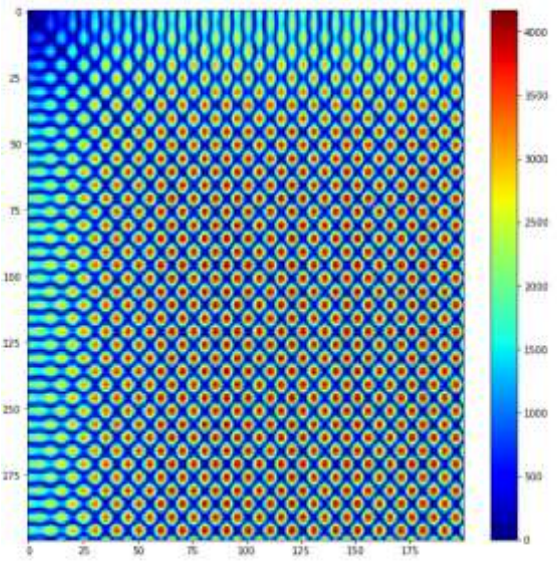
The python code used on the time series to create the recurrence plots is presented bellow.

```
def recurrence_plot(s, eps=None, steps=None):
    if eps==None: eps=0.1
    d = sk.metrics.pairwise.pairwise_distances(s)
    d = np.floor(d / eps)
    return d
```

- input **s:** is the time history represented as a matrix with dimensions (sequence length , number of variables). In the example (8000,2).
- optional parameter **eps:** is a normalization value
- output **d**: is a matrix with dimensions (sequence length, sequence length), where $d_{i,j}$ is the euclidian distance between the values of the time-series at time i and j.

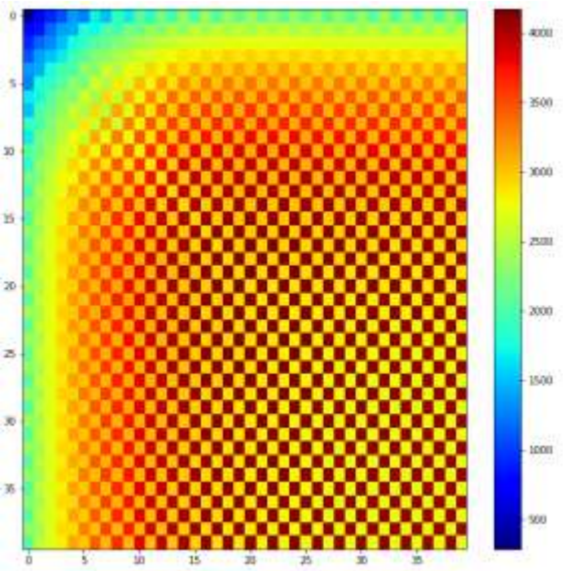
$$d_{i,j} = \frac{\sqrt{(s_{i,1} - s_{j,1})^2 + (s_{i,2} - s_{j,2})^2}}{eps}$$

The result of this process for eps=0.001, visualized as a colour map, is:



Step 2 Down-sampling

The processing of the above data is computationally prohibiting. In order to process the images, down-sampling is necessary. This was achieved by grouping together the values in a particular area of the plot and using the maximum value of the area, as a singular value. For our example, we have rows=8000 and columns= 8000 values and want to down-sample to rows=200, columns=200. We group together the values every 8000/200=40 rows and 40 columns and use the maximum value for each "square".

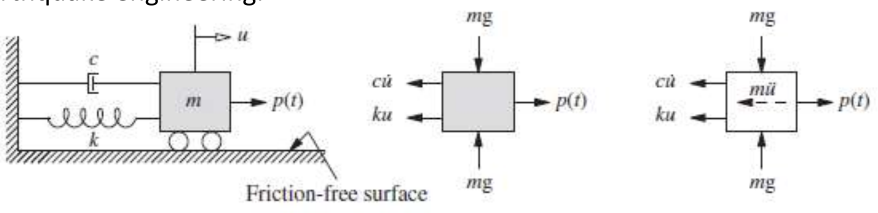


6 Application in simple engineering problems

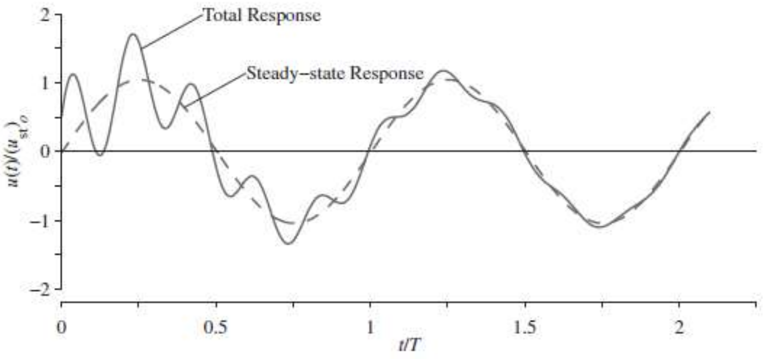
In this chapter, using the concepts that were presented in the previous chapters we will try to apply neural networks in some simple engineering problems. The focus in this paper will be on supervised classification and regression of sequences(time-series).

6.1 Detecting Resonance in Harmonic Oscillations

The first engineering problem we will try to solve is detecting resonance on a single degree of freedom oscillator subjected to various harmonic excitations $p(t)$. The response of SDF systems to harmonic excitation is a classical topic in structural dynamics, not only because such excitations are encountered in engineering systems (e.g., force due to unbalanced rotating machinery), but also because understanding the response of structures to harmonic excitation provides insight into how the system will respond to other types of forces. Furthermore, the theory of forced harmonic vibration has several useful applications in earthquake engineering.



The response to harmonic vibrations with viscous damping is comprised of two parts. The steady-state response and transient response.



The total displacement can be calculated as:

$$u(t) = u_{transient}(t) + u_{steady-state}(t)$$
$$u_{transient}(t) = e^{-\zeta\omega_n t}(A * cos(\omega_D t) + B * sin(\omega_D t))$$
$$u_{steady-state}(t) = C * cos(\omega t) + D * sin(\omega t)$$

Where

ω is the forced vibration frequency

ω_n is the natural system frequency

$\omega_D = \omega_n\sqrt{1-\zeta^2}$ is the damped system frequency

A, B constants determined by the initial conditions

C, D constants determined by the forced vibration

The steady-state dynamic response, a sinusoidal oscillation at the forcing frequency, may be expressed as:

$$u_{steady-state}(t) = u_0 * \sin(\omega * t - \varphi) = (u_{st})_0 * R_d * \sin(\omega * t - \varphi)$$

Where

$u_0 = \sqrt{C^2 + D^2}$: the response amplitude

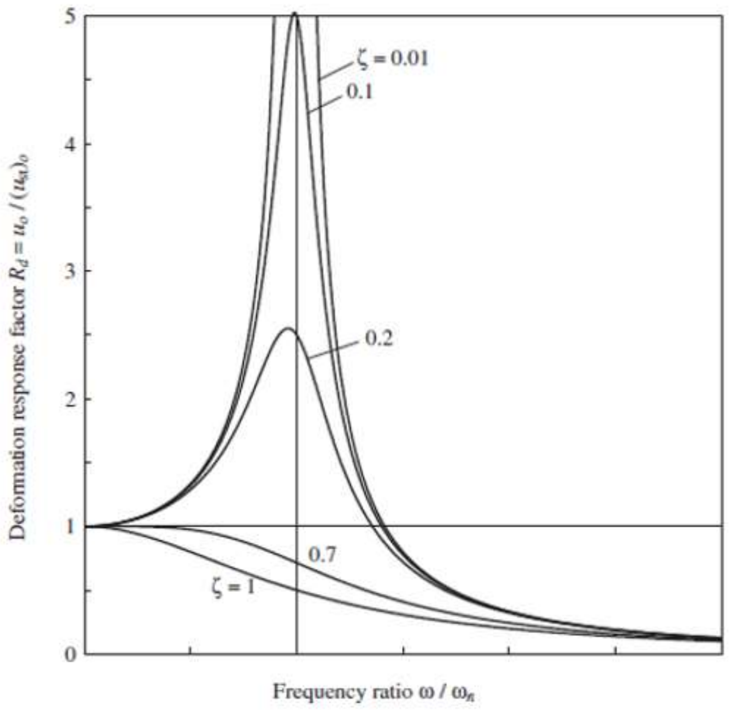
R_d : **deformation response factor.**

$(u_{st})_0$: the maximum static deformation, ignoring the dynamic effects signified by the acceleration term

$\varphi = tan^{-1}(-D/C)$: phase angle

The deformation response factor is directly connected to the maximum deformation and can be calculated as:

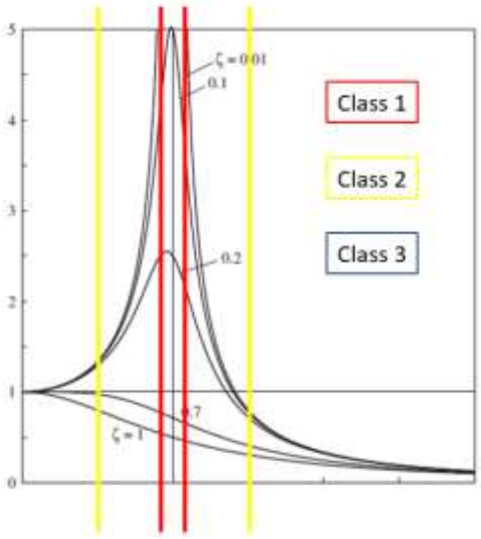
$$R_d = \frac{1}{\sqrt{[1-(\omega/\omega_n)^2]^2 + [2 * \omega * \zeta * (\omega/\omega_n)]^2}}$$



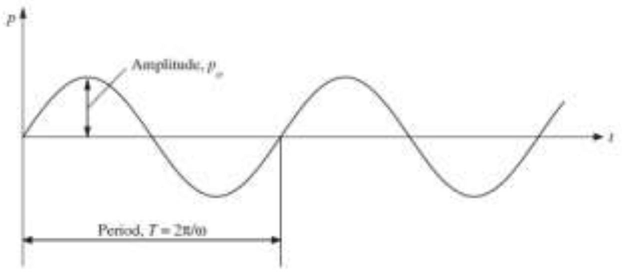
A more in depth explanation is given in chapter 3 of "Dynamics of Structures" by Anil K. Chopra.

Task

In this example the algorithm will identify the cases, which the system under a harmonic excitation approaches resonance, given as input the harmonic force and the resulting displacement, velocity and acceleration. More specifically, when $0{,}9 < \frac{\omega}{\omega_n} < 1.1$, where the maximum deformation of the harmonic vibration is at its highest. Furthermore, we will try to also differentiate the cases which $0.5 < \frac{\omega}{\omega_n} < 1.5$.



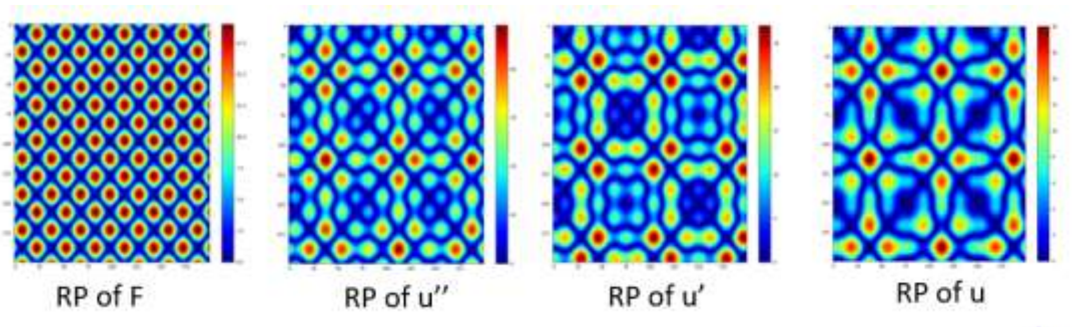
- Oscillators were defined with natural periods: 0.5s, 1.5s, 2.5s, 3.5s, 4.5s, 5.5s, 6.5s, 7.5s, 8.5s, 9.5s.
- For each of those the damping ratio was determined as: 0%, 10%, 20%, 30%, 40%, 50%. In total 600 sdof systems.
- The harmonic excitations had the format $p(t) = p_0 * \sin(\omega * t)$, where p_0 is the amplitude or maximum value of the force and ω the exciting frequency or forcing frequency. The amplitude had values: 10kN, 50kN, 90kN, 130kN, 170kN. The forced frequency: 0.5s, 1.5s, 2.5s, 3.5s, 4.5s, 5.5s, 6.5s, 7.5s, 8.5s, 9.5s. In total 50 harmonic excitations.



- Every system was subjected to every excitation. This resulted in 3000 harmonic vibrations or samples.

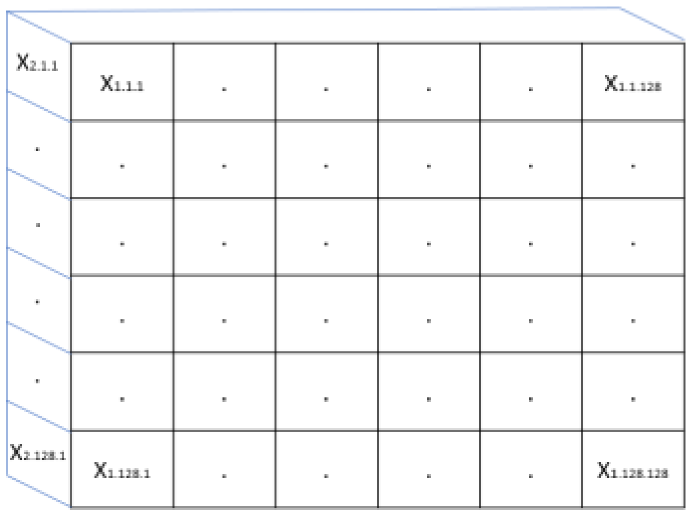
Approaches

The first model is a simple architecture, inspired by Lenet, with 1 dimensional convolutions. The second and third models try to improve the already satisfactory accuracy with recurrence plots, which were introduced in chapter 5. The difference between them is that in the second we down-sample the RP from 200 rows and columns to 40.
Overall, this proved a relatively easy task that achieved accuracies close to 100%. Transforming the data with recurrence plots seemed to improve the accuracy, but was computationally difficult and likely impossible with more complicated tasks and data. Down-sampling helped significantly in this regard, but dropped the accuracy to bellow that of the default 1 dimensional convolution.
It should be noted that for the recurrence plots **we treated each variable as a univariate time-series** and created a recurrence plot for each separately.



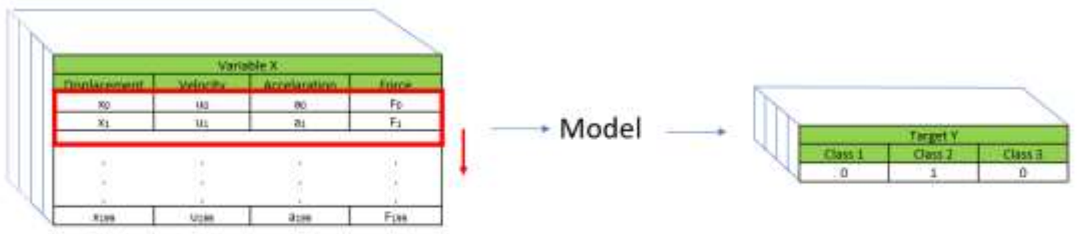
As input to the neural network we used a matrix with dimensions (number of recurrence plots , rows, columns).

Input X (One sample)

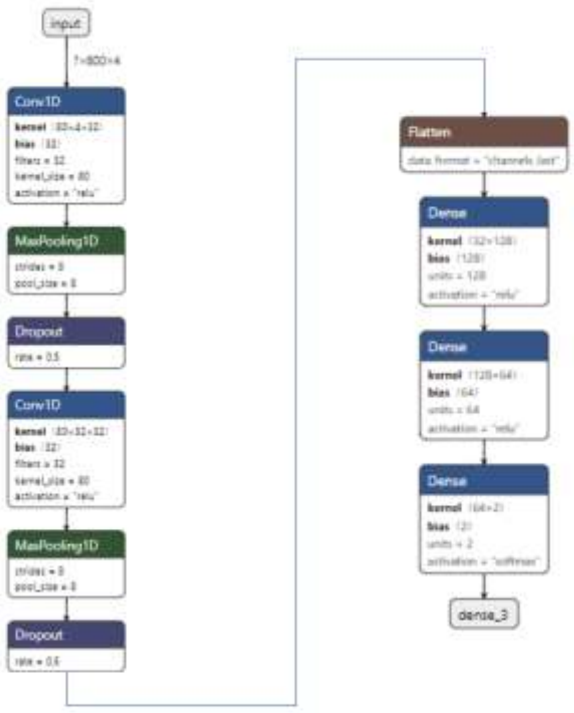


6.1.1 Model 1

Model one uses 1-dimensional convolution layers and an architecture similar to lenet.

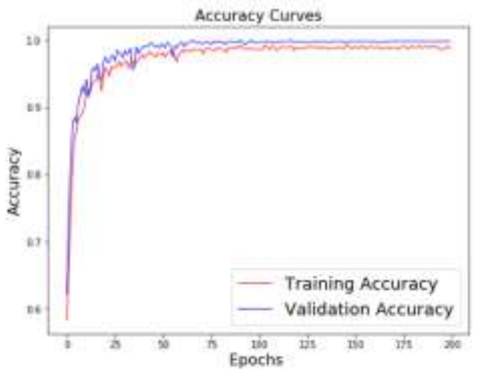


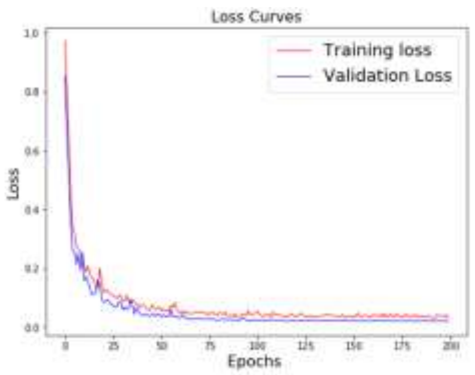
Architecture and hyperparameters



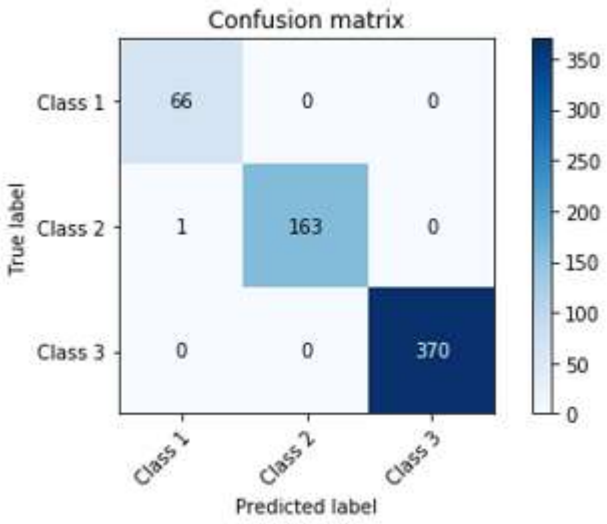
Loss Function	**Categorical Crossentropy**
Optimizer	**'Adams' optimizer with initial learning rate 0.001, divided by 2 every 10 epochs**
Other parameters	• **Batch size=100** • **Epochs=200** • **Early Stoppage (patience 50)** • **Test set split=20%**

Results



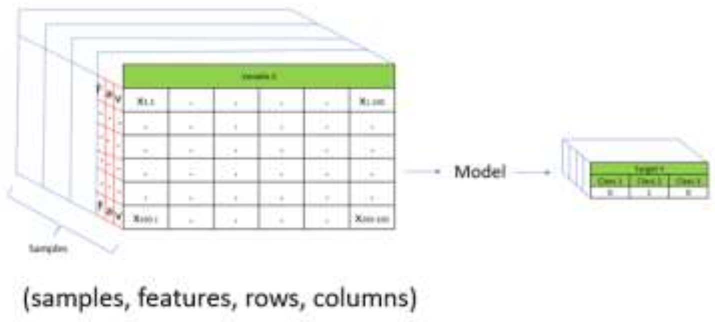


Confusion Matrix

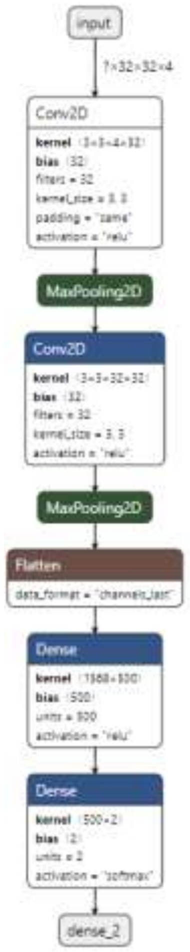


6.1.2 Model 2

Each feature of each time series was converted to a 200 by 200 RP. This resulted in an input with dimensions (samples, 4 features, 200 rows, 200 columns).

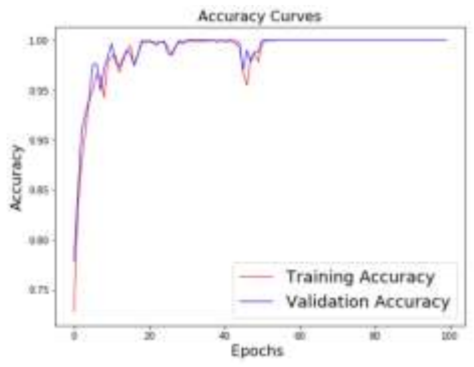


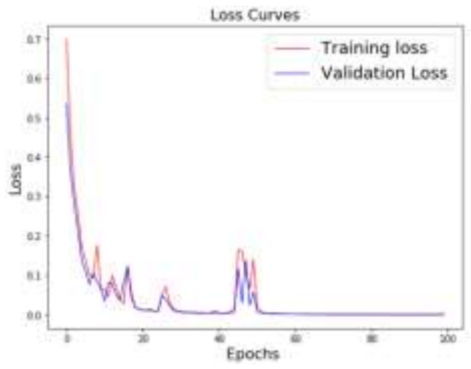
Architecture and hyperparameters



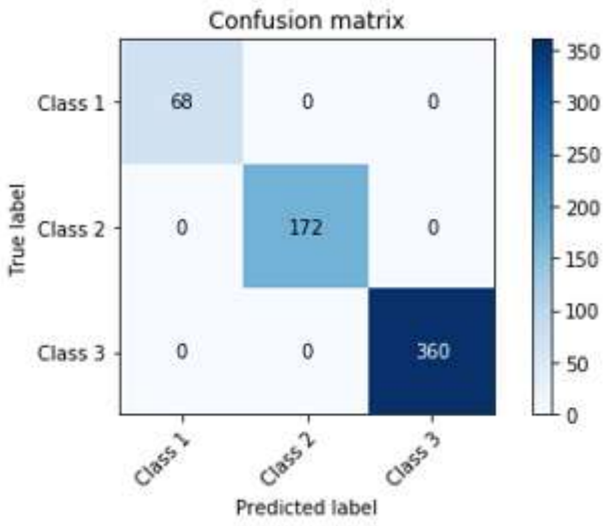
Loss Function	**Categorical Crossentropy**
Optimizer	**'Adams' optimizer with initial learning rate 0.001, divided by 2 every 10 epochs**
Other parameters	• **Batch size=100** • **Epochs=2000-Early Stoppage (patience 50)** • **Test set split=20%**

Results



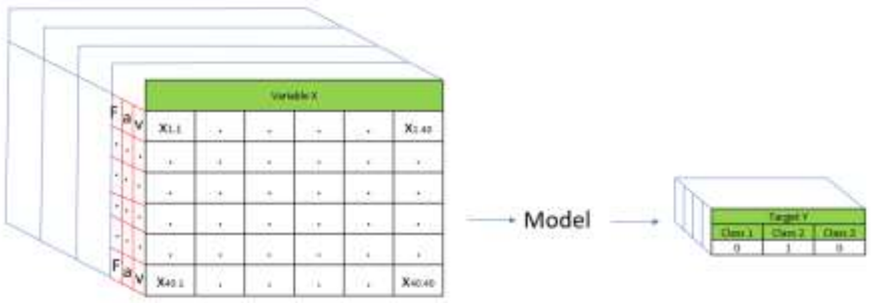


Confusion Matrix

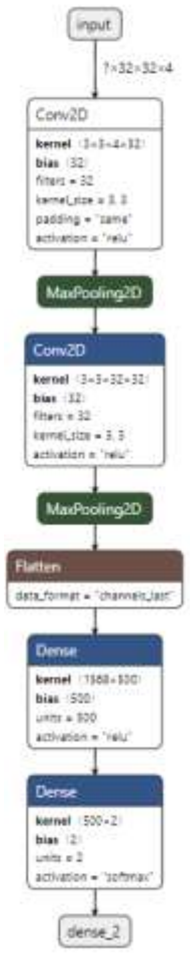


6.1.3 Model 3

Each feature of each time series was converted to a 200 by 200 RP. Then down-sampled to 40 by 40. This resulted in an input with dimensions (samples, 4 features, 40 rows, 40 columns).

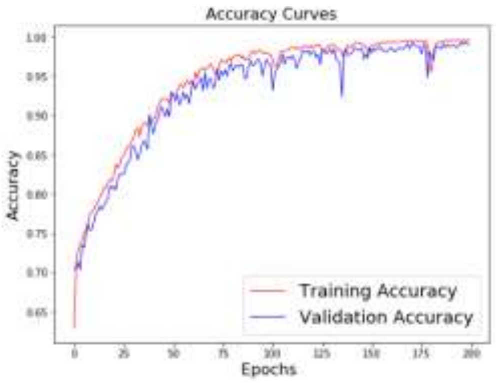


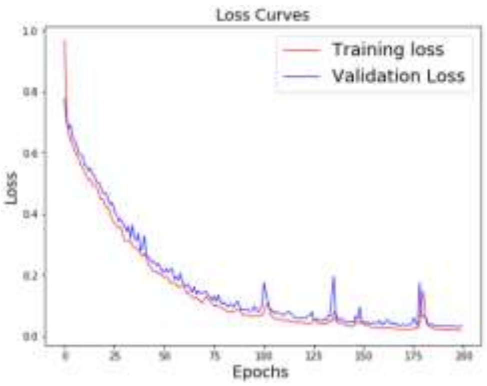
Architecture and hyperparameters



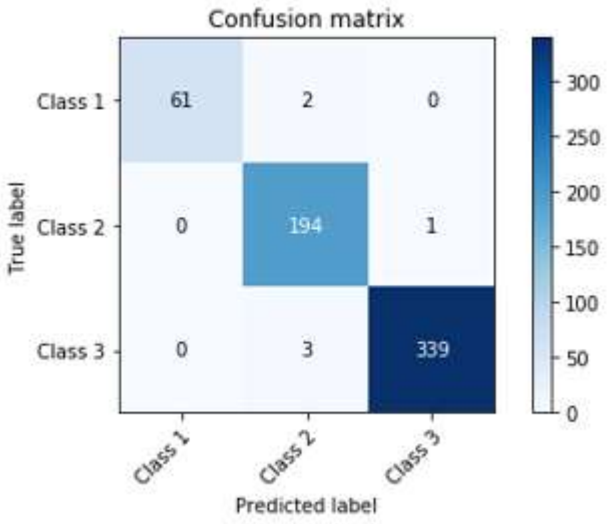
Loss Function	**Categorical Crossentropy**
Optimizer	**‘Adams’ optimizer with initial learning rate 0.001, divided by 2 every 10 epochs**
Other parameters	• **Batch size=100** • **Early Stoppage (patience 50)** • **Test set split=20%**

Results





Confusion Matrix



	Class 1	Class 2	Class 3
Class 1	61	2	0
Class 2	0	194	1
Class 3	0	3	339

6.1.4 Classifying Forced Oscillations by Damping

Ground motions

For the second example several ground motions were selected from NGA Database from PEER Berkley.

- Most of the motions suggested in FEMA 440a were used, as well as others
- Only motions with time steps of 0.005, 0.01, 0.02 were used. This was done because in order to improve accuracy all results were converted to have the same time step of 0.005. This was performed easier in these cases.
- Each motion had acceleration grams for three directions all of which were used as separate motions

In the end 141 ground motions with durations of 40 seconds and a timestep of 0.005 seconds were used.

Analysis

For this example, in order to calculate the response spectrum of the SDOF oscillation, linear analyses were performed using Newmark-Beta method as described in 'Dynamics of Structures' by Anil K. Chopra. In this example the constant average acceleration variant was used.

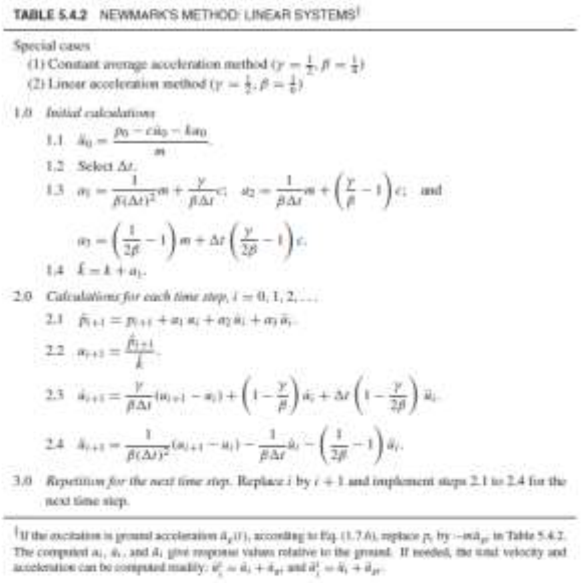


TABLE 5.4.2 NEWMARK'S METHOD: LINEAR SYSTEMS[†]

Special cases
(1) Constant average acceleration method ($\gamma = \frac{1}{2}, \beta = \frac{1}{4}$)
(2) Linear acceleration method ($\gamma = \frac{1}{2}, \beta = \frac{1}{6}$)

1.0 *Initial calculations*

1.1 $\ddot{u}_0 = \dfrac{p_0 - c\dot{u}_0 - ku_0}{m}$.

1.2 Select Δt.

1.3 $a_1 = \dfrac{1}{\beta(\Delta t)^2}m + \dfrac{\gamma}{\beta \Delta t}c$; $a_2 = \dfrac{1}{\beta \Delta t}m + \left(\dfrac{\gamma}{\beta} - 1\right)c$; and

$a_3 = \left(\dfrac{1}{2\beta} - 1\right)m + \Delta t\left(\dfrac{\gamma}{2\beta} - 1\right)c$.

1.4 $\hat{k} = k + a_1$.

2.0 *Calculations for each time step, $i = 0, 1, 2, \ldots$*

2.1 $\hat{p}_{i+1} = p_{i+1} + a_1 u_i + a_2 \dot{u}_i + a_3 \ddot{u}_i$.

2.2 $u_{i+1} = \dfrac{\hat{p}_{i+1}}{\hat{k}}$.

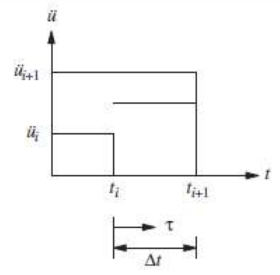
2.3 $\dot{u}_{i+1} = \dfrac{\gamma}{\beta \Delta t}(u_{i+1} - u_i) + \left(1 - \dfrac{\gamma}{\beta}\right)\dot{u}_i + \Delta t\left(1 - \dfrac{\gamma}{2\beta}\right)\ddot{u}_i$.

2.4 $\ddot{u}_{i+1} = \dfrac{1}{\beta(\Delta t)^2}(u_{i+1} - u_i) - \dfrac{1}{\beta \Delta t}\dot{u}_i - \left(\dfrac{1}{2\beta} - 1\right)\ddot{u}_i$.

3.0 *Repetition for the next time step.* Replace i by $i + 1$ and implement steps 2.1 to 2.4 for the next time step.

[†]If the excitation is ground acceleration $\ddot{u}_g(t)$, according to Eq. (1.7.6), replace p_i by $-m\ddot{u}_{gi}$ in Table 5.4.2. The computed u_i, $\dot{u}_i$, and $\ddot{u}_i$ give response values relative to the ground. If needed, the total velocity and acceleration can be computed readily: $\dot{u}_i^t = \dot{u}_i + \dot{u}_{gi}$ and $\ddot{u}_i^t = \ddot{u}_i + \ddot{u}_{gi}$.

Constant Average Acceleration



$$\ddot{u}(\tau) = \frac{1}{2}(\ddot{u}_{i+1} + \ddot{u}_i)$$

$$\dot{u}(\tau) = \dot{u}_i + \frac{\tau}{2}(\ddot{u}_{i+1} + \ddot{u}_i)$$

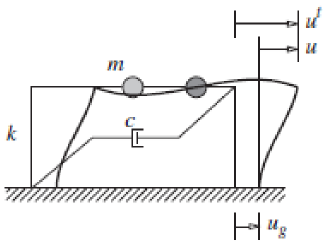
$$\dot{u}_{i+1} = \dot{u}_i + \frac{\Delta t}{2}(\ddot{u}_{i+1} + \ddot{u}_i)$$

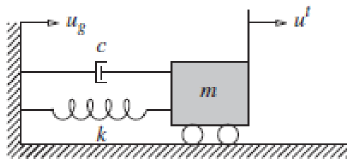
$$u(\tau) = u_i + \dot{u}_i\tau + \frac{\tau^2}{4}(\ddot{u}_{i+1} + \ddot{u}_i)$$

$$u_{i+1} = u_i + \dot{u}_i\,\Delta t + \frac{(\Delta t)^2}{4}(\ddot{u}_{i+1} + \ddot{u}_i)$$

Task

In this example, the task of the model will be to identify the damping of the oscillator given the applied external force and the resulting displacement, velocity and acceleration.



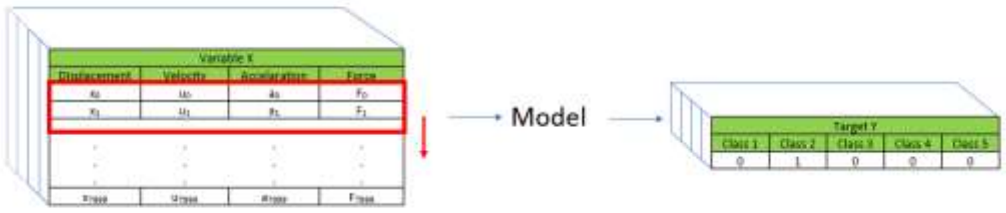


- 10 SDF systems where defined with masses and stiffnesses resulting in natural periods of systems: 0.5s, 1.5s, 2.5s, 3.5s, 4.5s, 5.5s, 6.5s, 7.5s, 8.5s, 9.5s.
- These systems were subjected to 141 ground motions.

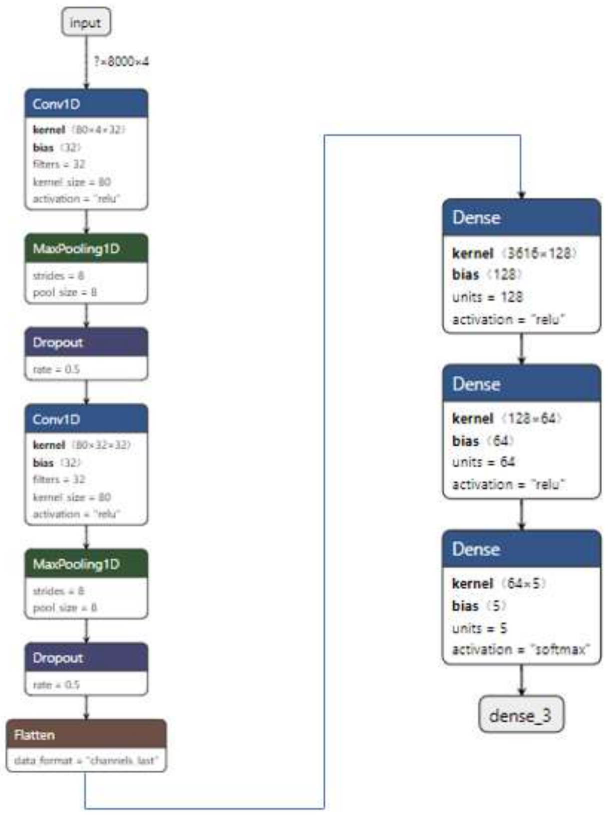
- Linear analysis were run with Newmark-beta method for damping ratios of 0%, 10%, 20%, 30% and 40%.
- This resulted in 7050 samples.

6.1.5 Model 1

As input we have a 3D matrix with dimensions (samples, features=4, sequence length=8000) and as output a one-hot distribution with 5 classes. In this model 1-dimensional CNN were used.

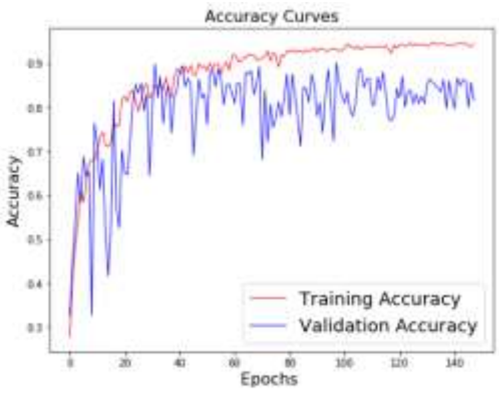


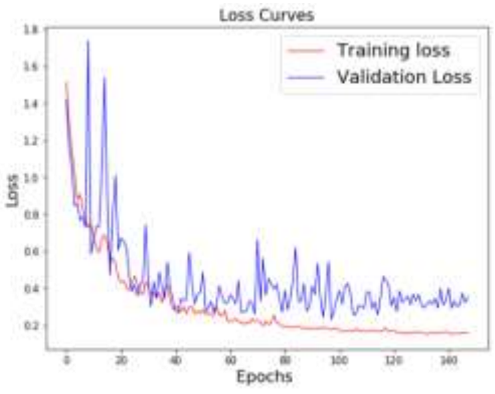
Architecture and hyperparameters



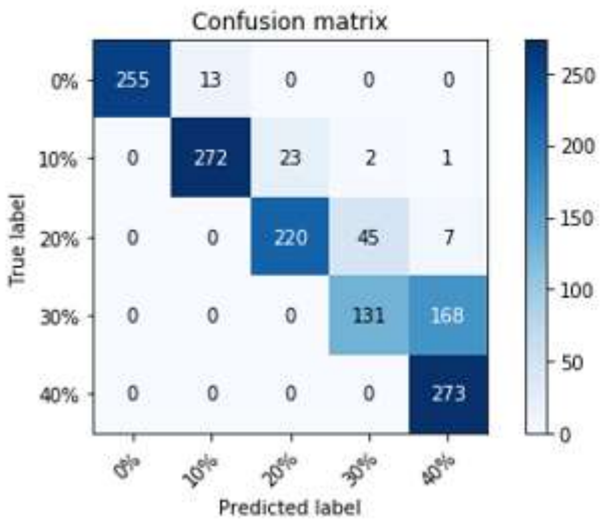
Loss Function	**Categorical Crossentropy**
Optimizer	**‘Adams’ optimizer with initial learning rate 0.001, divided by 2 every 10 epochs**
Other parameters	• **Batch size=100** • **Epochs=200 -Early Stoppage (patience 50)** • **Test set split=20%**

Results



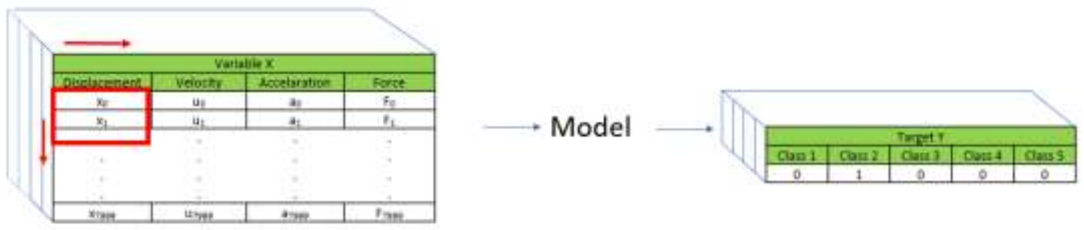


Confusion Matrix

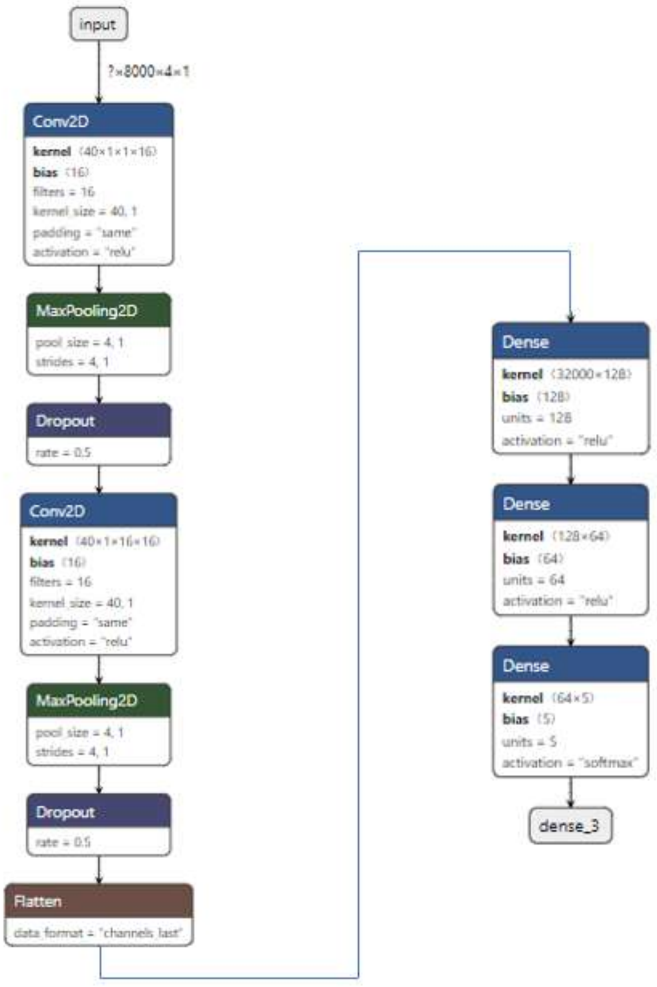


6.1.6 Model 2

The third model is similar to the previous one but uses 2-dimensional convolution layers.

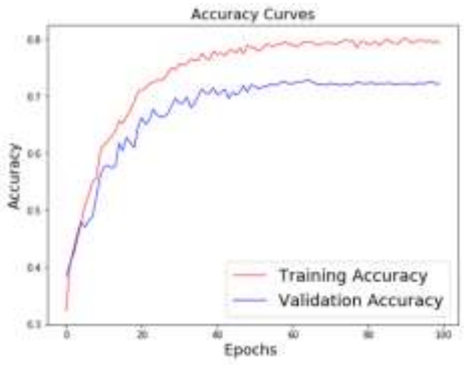


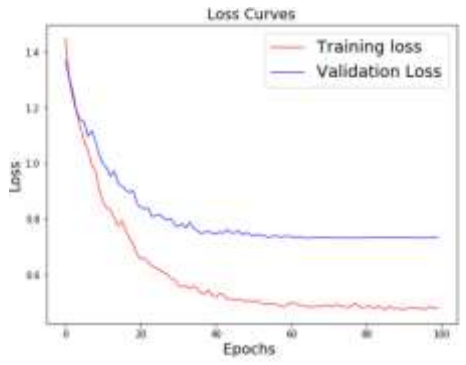
Architecture and hyperparameters



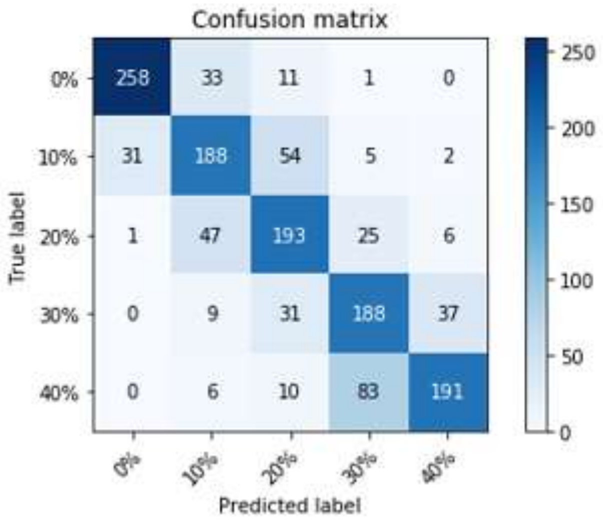
Loss Function	**Categorical Crossentropy**
Optimizer	**‘Adams’ optimizer with initial learning rate 0.001, divided by 2 every 10 epochs**
Other parameters	• **Batch size=100** • **Epochs=100 -Early Stoppage (patience 50)** • **Test set split=20%**

Results





Confusion Matrix



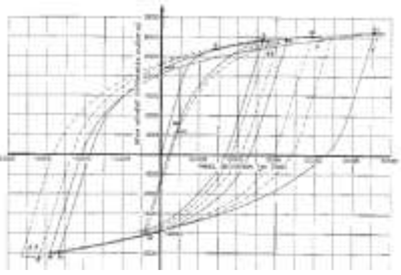
6.1.7 Nonlinear Systems Classification

As previously mentioned, the equation of motion is:

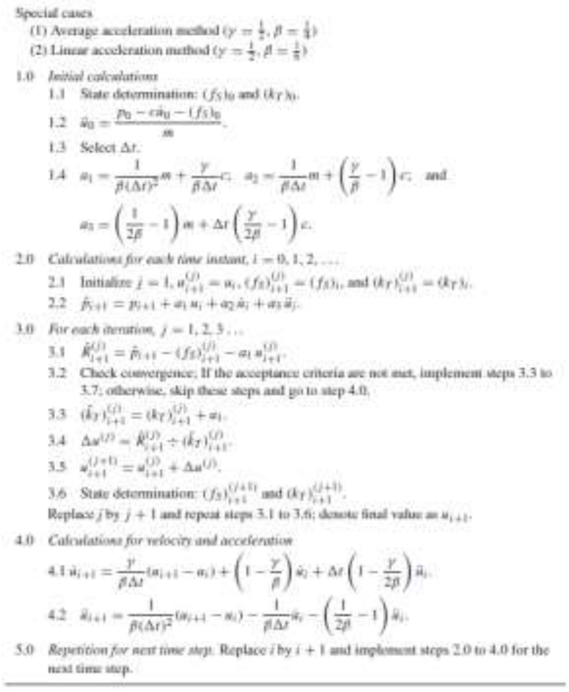
$$m * \ddot{u} + c * \dot{u} + fs(u) = p(t)$$

subject to the initial conditions: $u_0 = u(0)$ $\dot{u}_0 = \dot{u}(0)$
In linear analysis, the resisting force was calculated as $fs(u) = k * u$. In nonlinear analysis the force-displacement relation is more complicated.

From 'Dynamics of Structures' by Anil K. Chopra.
"Since the 1960s hundreds of laboratory tests have been conducted to determine the force– deformation behavior of structural components for earthquake conditions. During an earthquake structures undergo oscillatory motion with reversal of deformation. Cyclic tests simulating this condition have been conducted on structural members, assemblages of members, reduced-scale models of structures, and on small full-scale structures. The experimental results indicate that the cyclic force– deformation behavior of a structure depends on the structural material and on the structural system. The force– deformation plots show hysteresis loops under cyclic deformations because of inelastic behavior". An example of a hysteresis loop is presented bellow.



Since the 1960s many computer simulation studies have focused on the earthquake response of SDF systems with their force–deformation behavior defined by idealized versions of experimental curves.
In this project, the nonlinear analysis were performed with **Newmark's average acceleration method** as presented in 'Dynamics of Structures' by Anil K. Chopra.



Special cases
(1) Average acceleration method ($\gamma = \frac{1}{2}, \beta = \frac{1}{4}$)
(2) Linear acceleration method ($\gamma = \frac{1}{2}, \beta = \frac{1}{6}$)

1.0 *Initial calculations*
1.1 State determination: $(f_S)_0$ and $(k_T)_0$.
1.2 $\ddot{u}_0 = \frac{p_0 - c\dot{u}_0 - (f_S)_0}{m}$.
1.3 Select Δt.
1.4 $a_1 = \frac{1}{\beta(\Delta t)^2}m + \frac{\gamma}{\beta \Delta t}c$; $a_2 = \frac{1}{\beta \Delta t}m + \left(\frac{\gamma}{\beta} - 1\right)c$; and
$a_3 = \left(\frac{1}{2\beta} - 1\right)m + \Delta t\left(\frac{\gamma}{2\beta} - 1\right)c$.

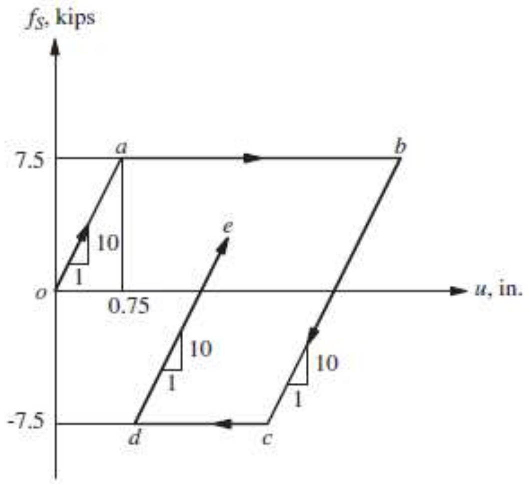
2.0 *Calculations for each time instant, $i = 0, 1, 2, \ldots$*
2.1 Initialize $j = 1$, $u_{i+1}^{(j)} = u_i$, $(f_S)_{i+1}^{(j)} = (f_S)_i$, and $(k_T)_{i+1}^{(j)} = (k_T)_i$.
2.2 $\hat{p}_{i+1} = p_{i+1} + a_1 u_i + a_2 \dot{u}_i + a_3 \ddot{u}_i$.

3.0 *For each iteration, $j = 1, 2, 3 \ldots$*
3.1 $\hat{R}_{i+1}^{(j)} = \hat{p}_{i+1} - (f_S)_{i+1}^{(j)} - a_1 u_{i+1}^{(j)}$.
3.2 Check convergence; If the acceptance criteria are not met, implement steps 3.3 to 3.7; otherwise, skip these steps and go to step 4.0.
3.3 $(\hat{k}_T)_{i+1}^{(j)} = (k_T)_{i+1}^{(j)} + a_1$.
3.4 $\Delta u^{(j)} = \hat{R}_{i+1}^{(j)} \div (\hat{k}_T)_{i+1}^{(j)}$.
3.5 $u_{i+1}^{(j+1)} = u_{i+1}^{(j)} + \Delta u^{(j)}$.
3.6 State determination: $(f_S)_{i+1}^{(j+1)}$ and $(k_T)_{i+1}^{(j+1)}$.
Replace j by $j + 1$ and repeat steps 3.1 to 3.6; denote final value as u_{i+1}.

4.0 *Calculations for velocity and acceleration*
4.1 $\dot{u}_{i+1} = \frac{\gamma}{\beta \Delta t}(u_{i+1} - u_i) + \left(1 - \frac{\gamma}{\beta}\right)\dot{u}_i + \Delta t\left(1 - \frac{\gamma}{2\beta}\right)\ddot{u}_i$.
4.2 $\ddot{u}_{i+1} = \frac{1}{\beta(\Delta t)^2}(u_{i+1} - u_i) - \frac{1}{\beta \Delta t}\dot{u}_i - \left(\frac{1}{2\beta} - 1\right)\ddot{u}_i$.

5.0 *Repetition for next time step.* Replace i by $i + 1$ and implement steps 2.0 to 4.0 for the next time step.

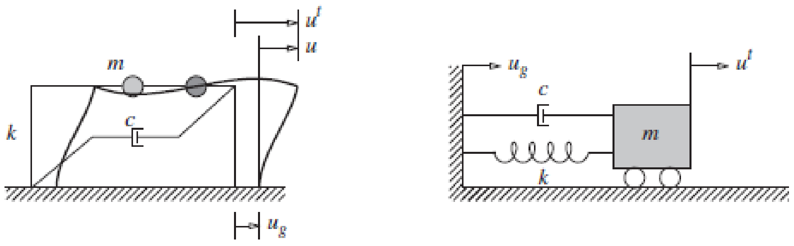
The response-deformation relation was approximated with **bilinear models**.



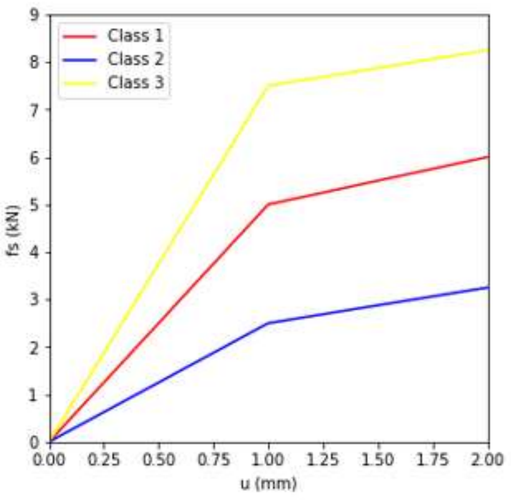
The “Python” code used is presented in appendix A.

Task

The task of the algorithm is to identify the sdof oscillators with different response force-displacement relations, subjected to ground motions. The input to the algorithm was the ground motion and the resulting acceleration, velocity and displacement of the oscillator.



- 10 SDF systems where defined with masses: 5kN, 10kN, 15kN, 20kN, 25kN, 30kN, 35kN, 40kN, 45kN, 50kN
- The damping ratio of each system was $\zeta = 5\%$. This resulted in damping coefficient $c = 2 * m * \zeta * \omega$, where $\omega = \sqrt{k\_el/m}$.
- These systems were subjected to 141 ground motions.
- Non-linear analysis were performed with Newmarks average acceleration method for the bilinear models presented bellow.



Model 1: $u_y = 1mm$ $Kel = 7500\frac{kN}{m}$ $Kpl = 0.1 * Kel$

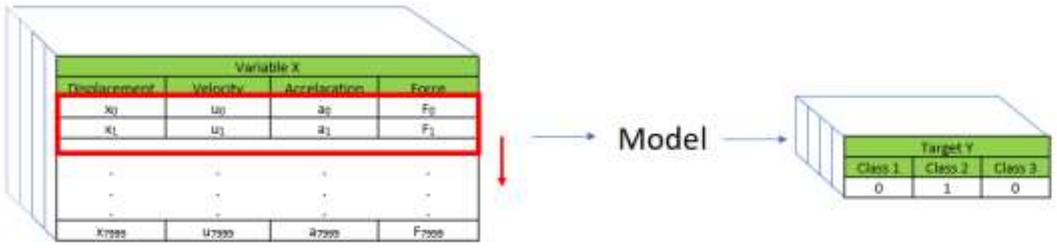
Model 2: $u_y = 1mm$ $Kel = 5000\frac{kN}{m}$ $Kpl = 0.2 * Kel$

Model 3: $u_y = 1mm$ $Kel = 2500\frac{kN}{m}$ $Kpl = 0.3 * Kel$

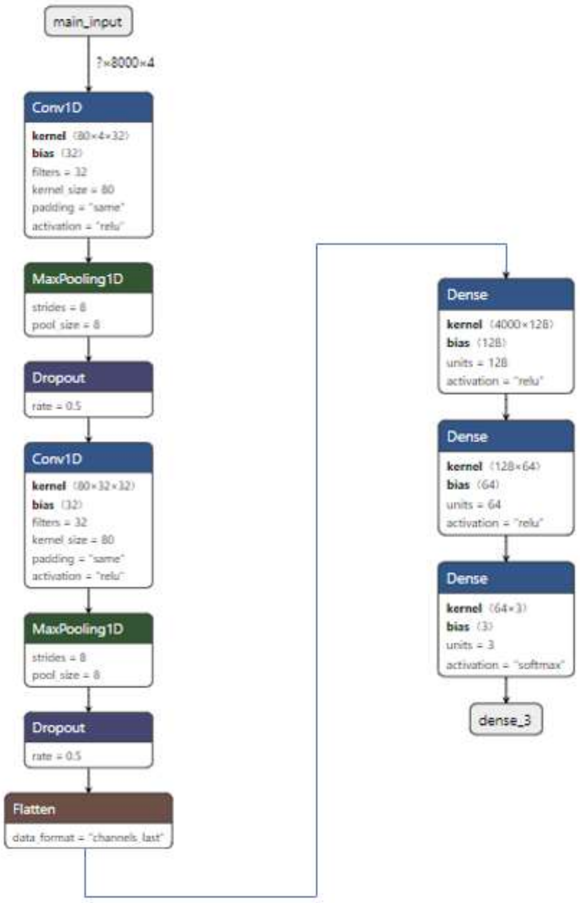
- This resulted in 4230 samples.

6.1.8 Model 1

The first model is based, again on 1 dimensional CNNs.

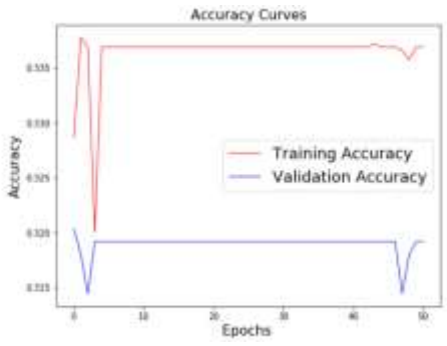


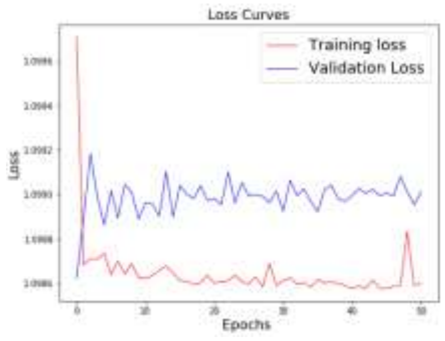
Model and Hyperparameters

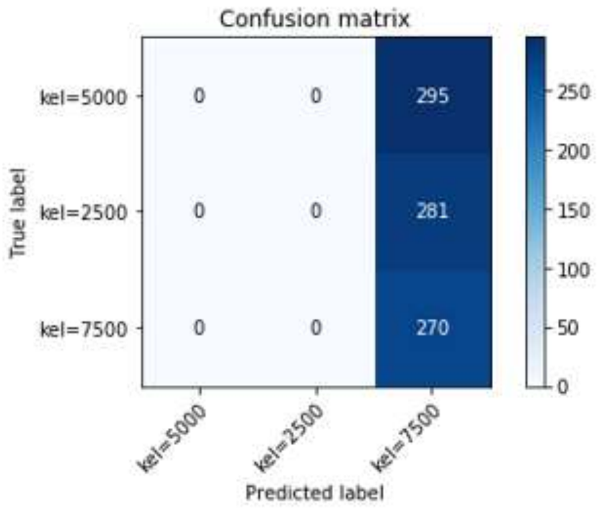


Loss Function	**Categorical Crossentropy**
Optimizer	**'Adams' optimizer with initial learning rate 0.001, divided by 2 every 10 epochs**
Other parameters	• **Batch size=100** • **Epochs=200 -Early Stoppage (patience 50)** • **Test set split=20%**

Results

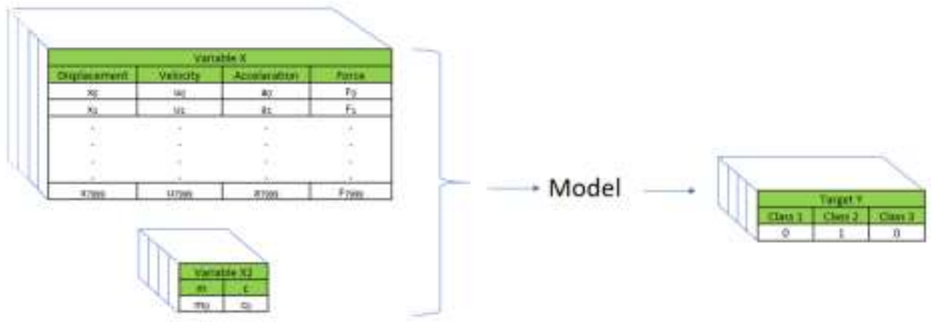

Accuracy Curves
Training Accuracy
Validation Accuracy
Accuracy
Epochs


Loss Curves
Training loss
Validation Loss
Loss
Epochs

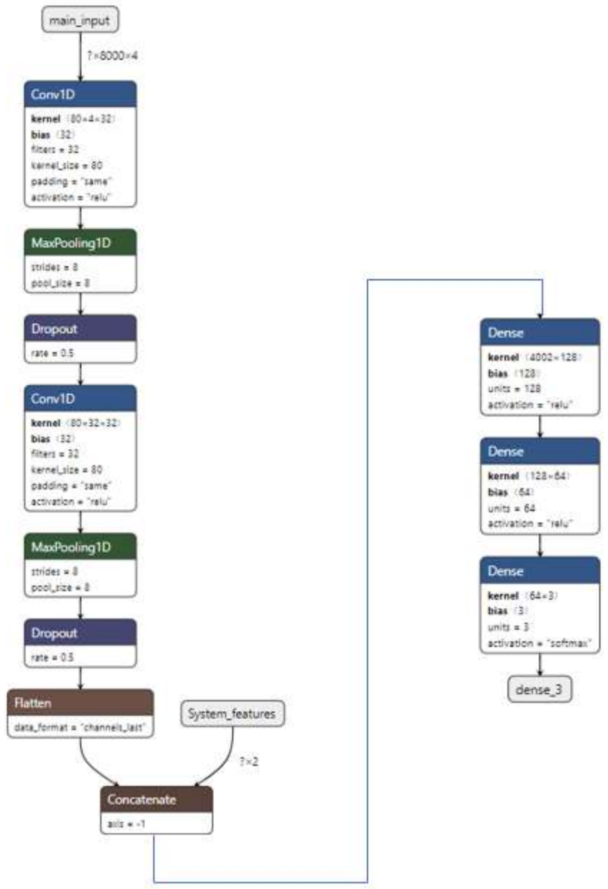

Confusion matrix
True label
Predicted label
kel=5000
kel=2500
kel=7500
0
0
295
0
0
281
0
0
270
250
200
150
100
50
0

6.1.9 Model 2

Since the first model couldn't converge to a solution at all from only the time-series, a second input to the algorithm was added with a concatenate layer from "Keras". The model runs 1-dimensional CNN on the time-series but before the output layer, accepts as input a 2d matrix with dimensions (samples, features=2), the features being the mass and damping coefficient of the system.

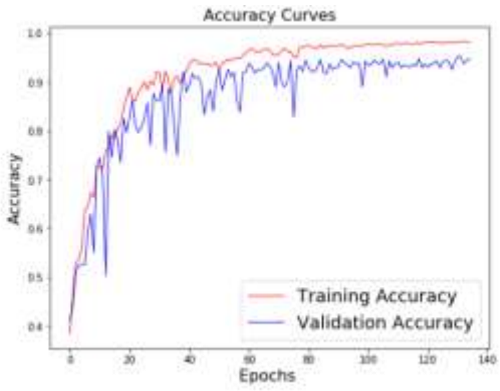


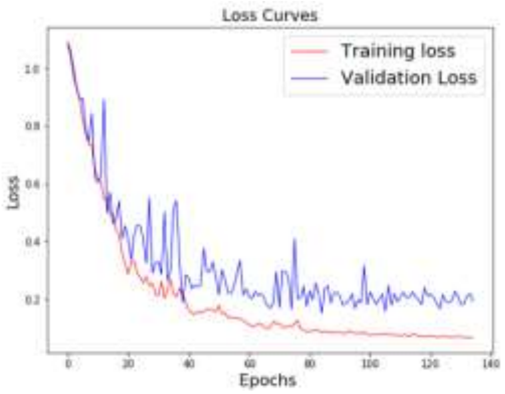
Model and Hyperparameters

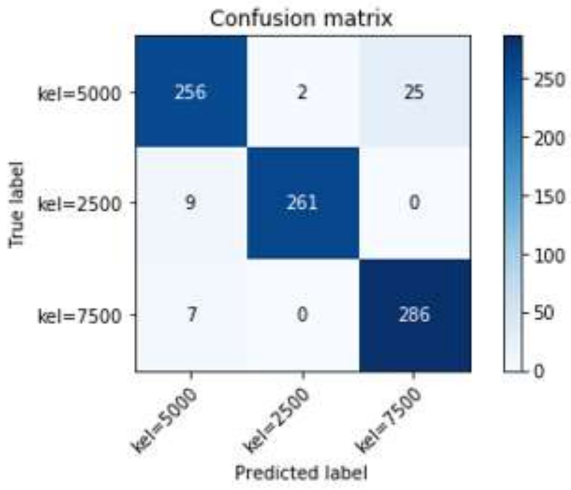


Loss Function	**Categorical Crossentropy**
Optimizer	**'Adams' optimizer with initial learning rate 0.001, divided by 2 every 10 epochs**
Other parameters	• **Batch size=100** • **Epochs=200 -Early Stoppage (patience 50)** • **Test set split=20%**

Results



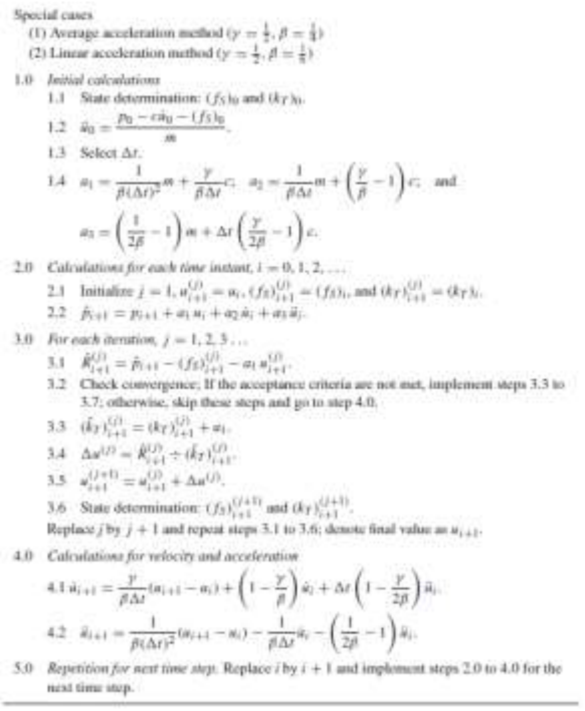




	kel=5000	kel=2500	kel=7500
kel=5000	256	2	25
kel=2500	9	261	0
kel=7500	7	0	286

6.2 Forecasting-Predicting

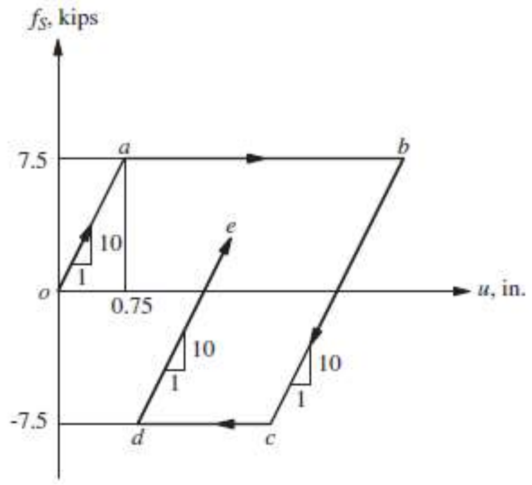
The non-linear analysis will be run with Newmark’s average acceleration method, same as the previous example.



Special cases
(1) Average acceleration method ($\gamma = \frac{1}{2}, \beta = \frac{1}{4}$)
(2) Linear acceleration method ($\gamma = \frac{1}{2}, \beta = \frac{1}{6}$)

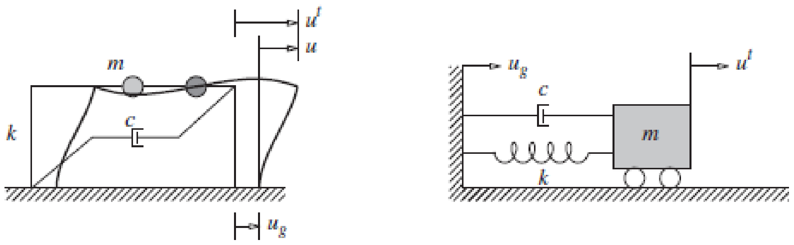
```
1.0 Initial calculations
    1.1 State determination: $(f_S)_0$ and $(k_T)_0$.
    1.2 $\ddot{u}_0 = \frac{p_0 - c\dot{u}_0 - (f_S)_0}{m}$.
    1.3 Select $\Delta t$.
    1.4 $a_1 = \frac{1}{\beta(\Delta t)^2}m + \frac{\gamma}{\beta \Delta t}c$;  $a_2 = \frac{1}{\beta \Delta t}m + \left(\frac{\gamma}{\beta} - 1\right)c$;  and
        $a_3 = \left(\frac{1}{2\beta} - 1\right)m + \Delta t\left(\frac{\gamma}{2\beta} - 1\right)c$.
2.0 Calculations for each time instant, $i = 0, 1, 2, \ldots$
    2.1 Initialize $j = 1$, $u_{i+1}^{(j)} = u_i$, $(f_S)_{i+1}^{(j)} = (f_S)_i$, and $(k_T)_{i+1}^{(j)} = (k_T)_i$.
    2.2 $\hat{p}_{i+1} = p_{i+1} + a_1 u_i + a_2 \dot{u}_i + a_3 \ddot{u}_i$.
3.0 For each iteration, $j = 1, 2, 3 \ldots$
    3.1 $\hat{R}_{i+1}^{(j)} = \hat{p}_{i+1} - (f_S)_{i+1}^{(j)} - a_1 u_{i+1}^{(j)}$.
    3.2 Check convergence; If the acceptance criteria are not met, implement steps 3.3 to
        3.7; otherwise, skip these steps and go to step 4.0.
    3.3 $(\hat{k}_T)_{i+1}^{(j)} = (k_T)_{i+1}^{(j)} + a_1$.
    3.4 $\Delta u^{(j)} = \hat{R}_{i+1}^{(j)} \div (\hat{k}_T)_{i+1}^{(j)}$.
    3.5 $u_{i+1}^{(j+1)} = u_{i+1}^{(j)} + \Delta u^{(j)}$.
    3.6 State determination: $(f_S)_{i+1}^{(j+1)}$ and $(k_T)_{i+1}^{(j+1)}$.
    Replace $j$ by $j + 1$ and repeat steps 3.1 to 3.6; denote final value as $u_{i+1}$.
4.0 Calculations for velocity and acceleration
    4.1 $\dot{u}_{i+1} = \frac{\gamma}{\beta \Delta t}(u_{i+1} - u_i) + \left(1 - \frac{\gamma}{\beta}\right)\dot{u}_i + \Delta t\left(1 - \frac{\gamma}{2\beta}\right)\ddot{u}_i$.
    4.2 $\ddot{u}_{i+1} = \frac{1}{\beta(\Delta t)^2}(u_{i+1} - u_i) - \frac{1}{\beta \Delta t}\dot{u}_i - \left(\frac{1}{2\beta} - 1\right)\ddot{u}_i$.
5.0 Repetition for next time step. Replace $i$ by $i + 1$ and implement steps 2.0 to 4.0 for the
    next time step.
```

The force-deformation relation was, again, approximated with **bilinear models**.

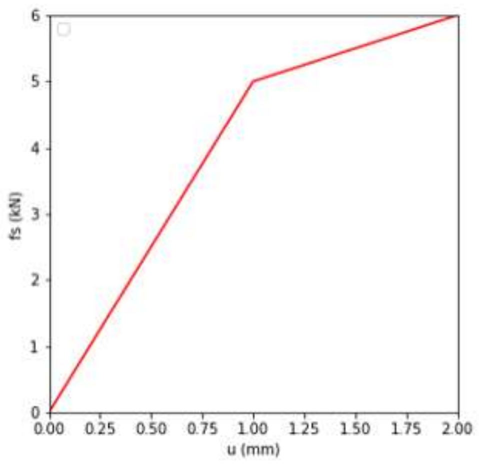


Task

In this example we will try to use LSTMs on regression tasks. The task of the algorithm is to forecast or predict the response of a sdof oscillator subjected to ground motions.



- A single SDF system was defined with mass 100kN, damping coefficient $c = 2 * m * \zeta * \omega$ and the deformation-response relation as depicted bellow.

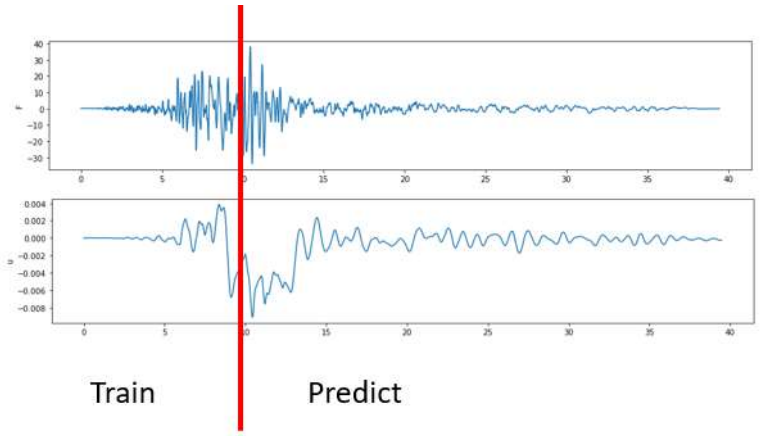


Model : $u_y = 1mm$ $\quad Kel = 5000\frac{kN}{m}$ $\quad Kpl = 0.2 * Kel$

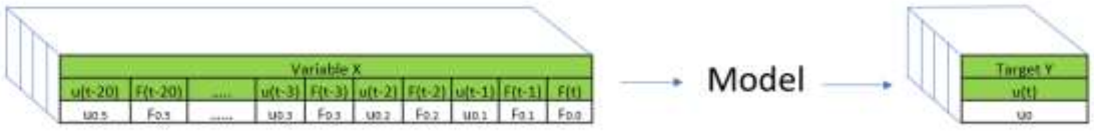
- The input and output are different for each model, since they are variations of a regression task and will be explained for each model separately.

6.2.1 Single Motion Forecasting

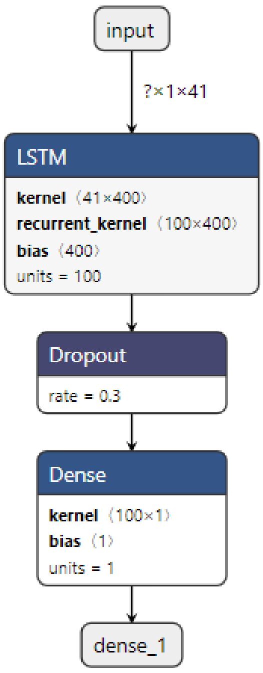
In this first example, the algorithm will be trained on the first seconds of a ground motion and asked to predict the response after, given the motion. The time series is converted to a supervised problem with the sliding window method presented in chapter 5



- As input we have the displacement and excitation force in the previous 20 steps and the excitation force in the current step. As output, we want the algorithm to predict the displacement at the current step.
- One method to express tasks with one sequence, that was used here, is to treat each step of the algorithm as one sample. This allows the output layer to be a simple linear fully connected layer with one neuron. Input dimensions (samples=sequence length, time step=1, features=11 values). Output dimensions (samples=sequence length, feature=1 value). It should be noted that tasks, expressed in this manner, shouldn't have the samples shuffled, as in all the other examples.
- In this particular task, the algorithms performance was improved from scaling the data. All the data was normalized (values between 0 and 1).
- The model was used in 2 separate ground motions.



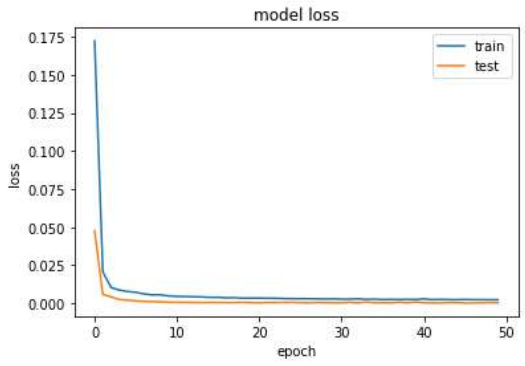
Model and Hyperparameters



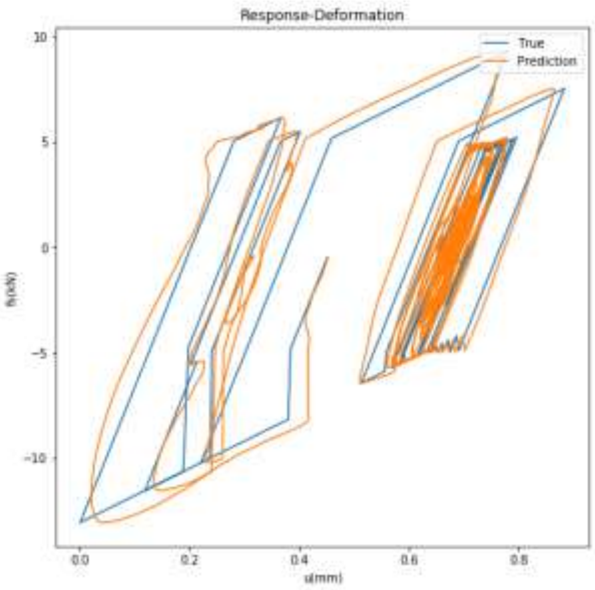
Loss Function	**Mean Squared Error**
Optimizer	**'Adams' optimizer with learning rate 0.001**
Other parameters	• **Batch size=Sequence length** • **Epochs=20** • **Shuffle=False**

Results for Ground Motion 1

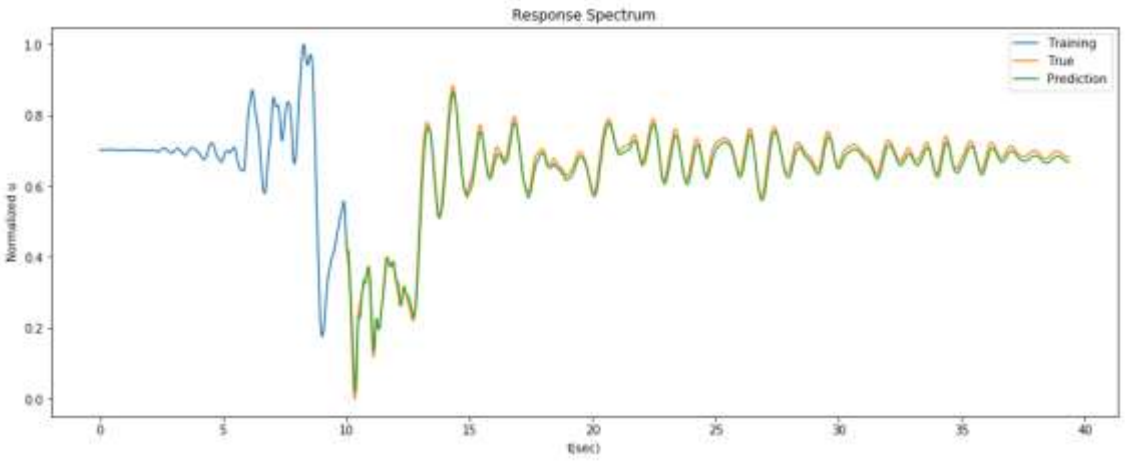
- Loss curve



- Predicted and True Hysteresis Loops

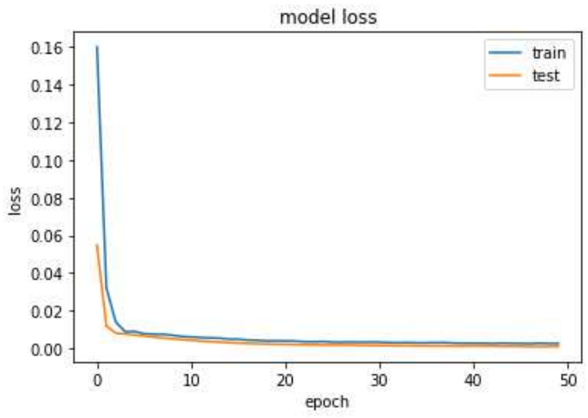


- Training, True and Predicted Response Spectrums

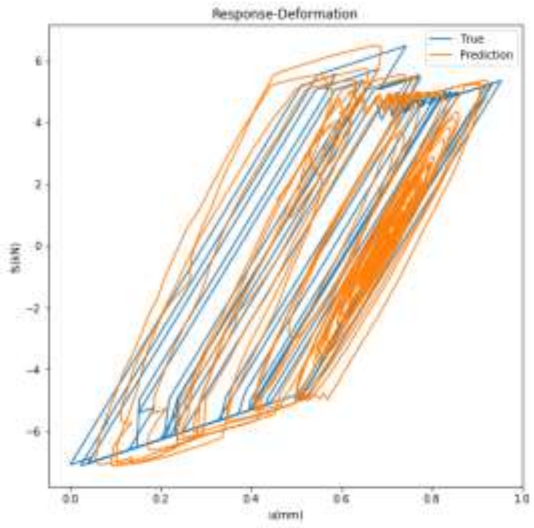


Results for Ground Motion 2

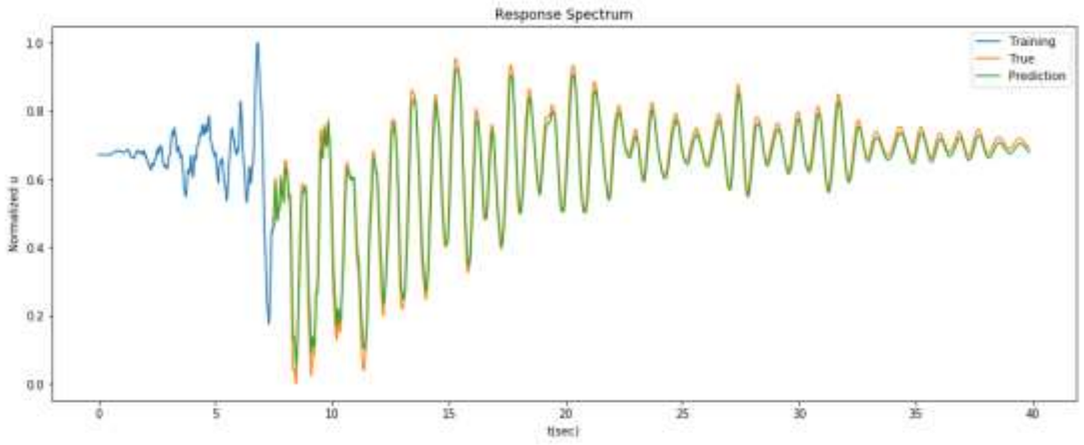
- Loss curve



- Predicted and True Hysteresis Loops



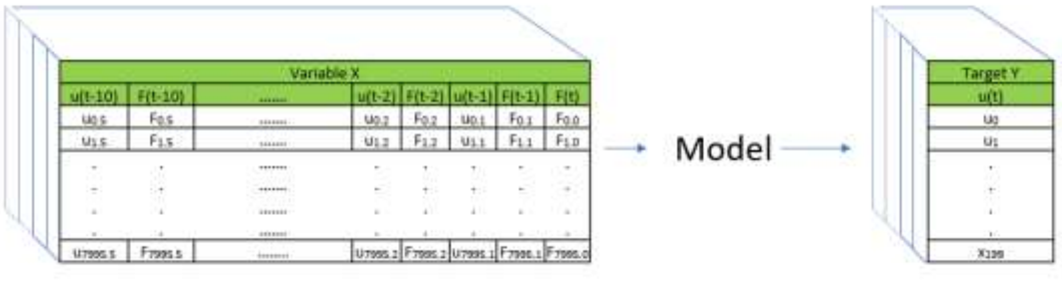
- Training, True and Predicted Response Spectrums



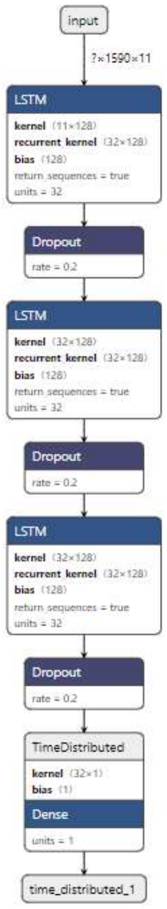
6.2.2 Forecasting

In this example the model will try to learn from multiple ground motions. The sliding window method was used again.

- The first 40 seconds of the ground motions were used with time step 0.005. The motions used included both ones with large displacements that pushed the sdof oscillator to the anelastic zone and milder ones that stayed elastic.
- As input we have the displacement and excitation force in the previous 10 steps and the excitation force in the current step. However, in this example in the sample axis we will have the ground motions, in the time axis the data at each step and the features will remain the same. Input dimensions (samples=141 ground motions, time=sequence length, features=11 values). As output, we want the algorithm to predict the displacement throughout each motion, so the dimensions will be (samples=141 ground motions, time=sequence length). This means that we cannot use the simple linear FC layer with 1 neuron as the output layer, like we did before. This was resolved with a time distributed FC layer which repeats the neuron for each time step and allows one more "Dimension". So the input will have dimensions (samples=ground motion, timestep=steps, features=21) and the output (samples=ground motions, timestep=steps)
- In this task, performance was improved again from scaling the data, so the data was normalized (values between 0 and 1).

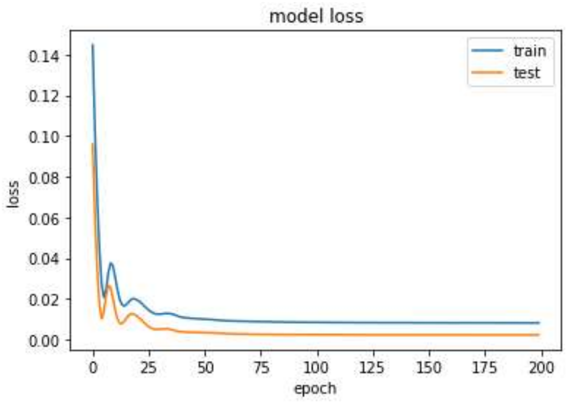


Model and Hyperparameters

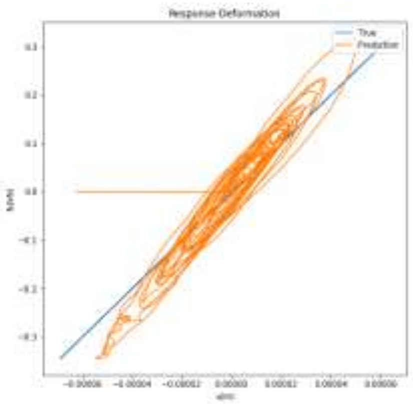


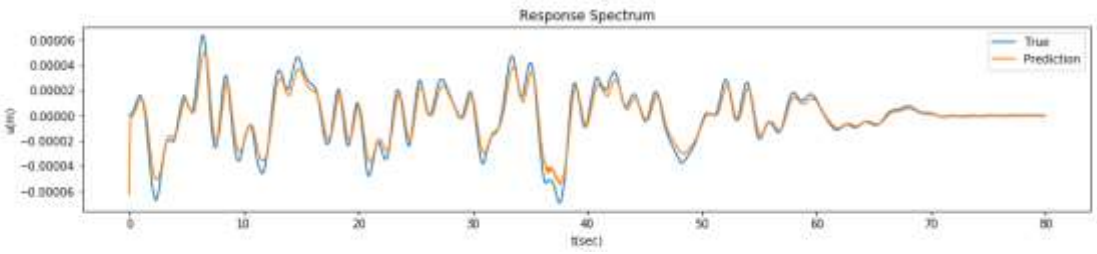
Loss Function	**Mean Squared Error**
Optimizer	**'Adams' optimizer with initial learning rate 0.001, divided by 2 every 10 epochs**
Other parameters	• **Batch size=50** • **Epochs=200** • **Test set split=20%**

Results

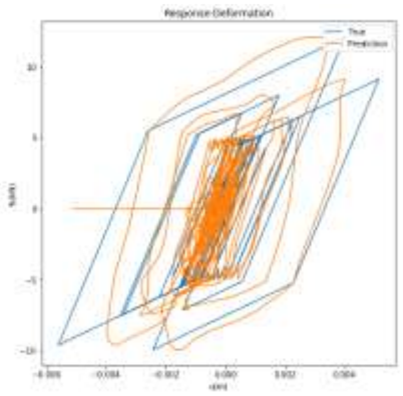


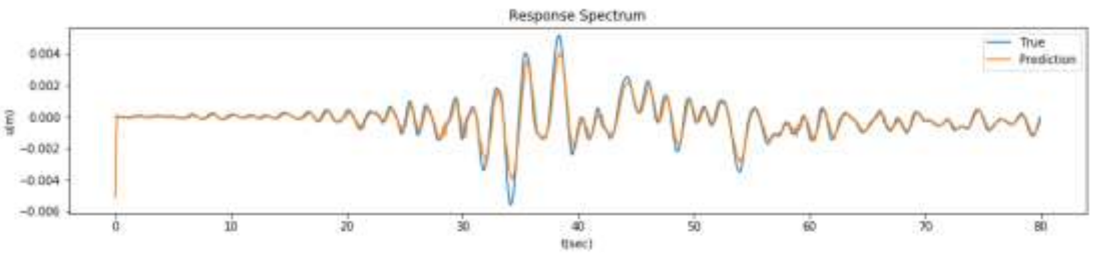
Test Motion 1





Test Motion 2

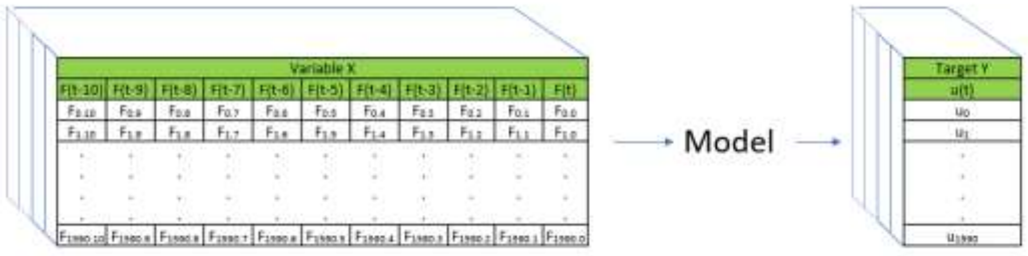




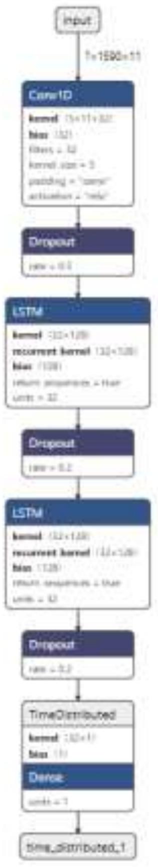
6.2.3 Prediction

In the previous model we used the values of the displacement in the previous time steps in order to train the algorithm for multiple ground motions. In this example, the model will attempt to detect the response spectrum of the non-linear analysis, having as input only the excitation.

- In previous examples, the ground motions were chosen randomly. This resulted in maximum displacements ranging from 1mm to 0.0001mm. This had a significant impact on the learning process, since the loss would remain small but the prediction would be significantly off. This was solved by scaling the data. The neural network, then, used the knowledge of the displacement at previous time steps to approximate the ones at the following time steps, thus “understanding” when the system crossed over to the inelastic part and the displacement was increased. In this task, since we do not use the previous displacements as inputs, we cant use scaling. In order to get good results, only motions over a certain intensity were used, so as not to interfere with the loss calculations. Also these motions were divided into sub-samples of 16 seconds, since lstm algorithms tend to lose efficiency at long sequences. In the end, all the sequences used had a length of 16 seconds with a time step of 0.005 second and a maximum displacement over 1mm.
- As input we have only the excitation force in the previous 10 steps and the excitation force in the current step. Input dimensions (samples= ground motions, time=sequence length, features). As output, we want the algorithm to predict the displacement throughout each motion, so the dimensions will be (samples=ground motions, time=sequence length), with a time distributed FC layer as the output layer.

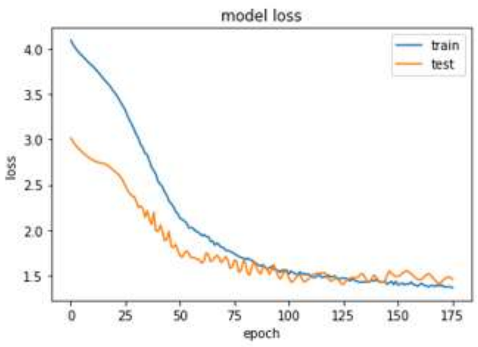

Variable X
F(t-10) F(t-9) F(t-8) F(t-7) F(t-6) F(t-5) F(t-4) F(t-3) F(t-2) F(t-1) F(t)
$F_{0,10}$ $F_{0,9}$ $F_{0,8}$ $F_{0,7}$ $F_{0,6}$ $F_{0,5}$ $F_{0,4}$ $F_{0,3}$ $F_{0,2}$ $F_{0,1}$ $F_{0,0}$
$F_{1,10}$ $F_{1,9}$ $F_{1,8}$ $F_{1,7}$ $F_{1,6}$ $F_{1,5}$ $F_{1,4}$ $F_{1,3}$ $F_{1,2}$ $F_{1,1}$ $F_{1,0}$
$F_{1980,10}$ $F_{1980,9}$ $F_{1980,8}$ $F_{1980,7}$ $F_{1980,6}$ $F_{1980,5}$ $F_{1980,4}$ $F_{1980,3}$ $F_{1980,2}$ $F_{1980,1}$ $F_{1980,0}$
Model
Target Y
u(t)
u_0
u_1
u_{1980}

Model and Hyperparameters

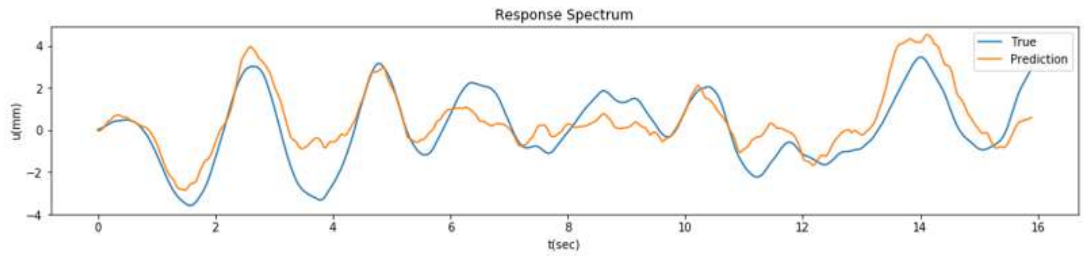


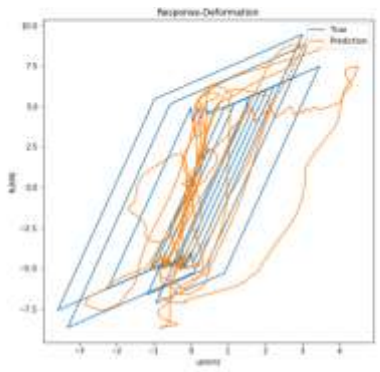
Loss Function	**Mean Squared Error**
Optimizer	**'Adams' optimizer with initial learning rate 0.001, divided by 2 every 10 epochs**
Other parameters	• **Batch size=50** • **Epochs=200** • **Test set split=20%**

Results

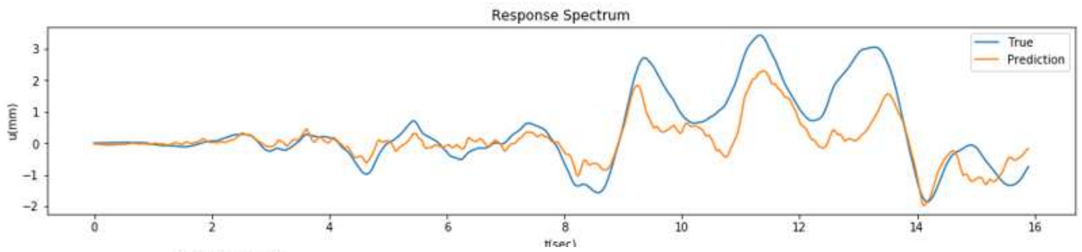


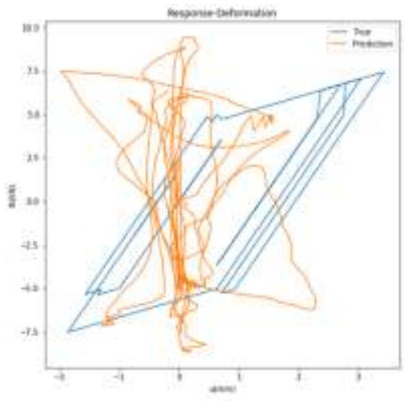
Test motion 1





Test motion 2





7 Sources

1. Goodfellow I., Bengio Y., Courville A. «Deep Learning».
2. Nishant Shukla «Machine Learning with Tensorflow».
3. Anil K. Chopra «Dynamics of Structures» (Fourth Edition).
4. John T. Katsikadelis «Dynamic Analysis of Structures».
5. https://www.kaggle.com/
6. https://machinelearningmastery.com/
7. https://ngawest2.berkeley.edu/
8. https://medium.com/
9. https://towardsdatascience.com/
10. Nima Hatami, Yann Gavet and Johan Debayle «Classification of Time-Series Images Using Deep Convolutional Neural Networks»

8 Appendix A Python Code

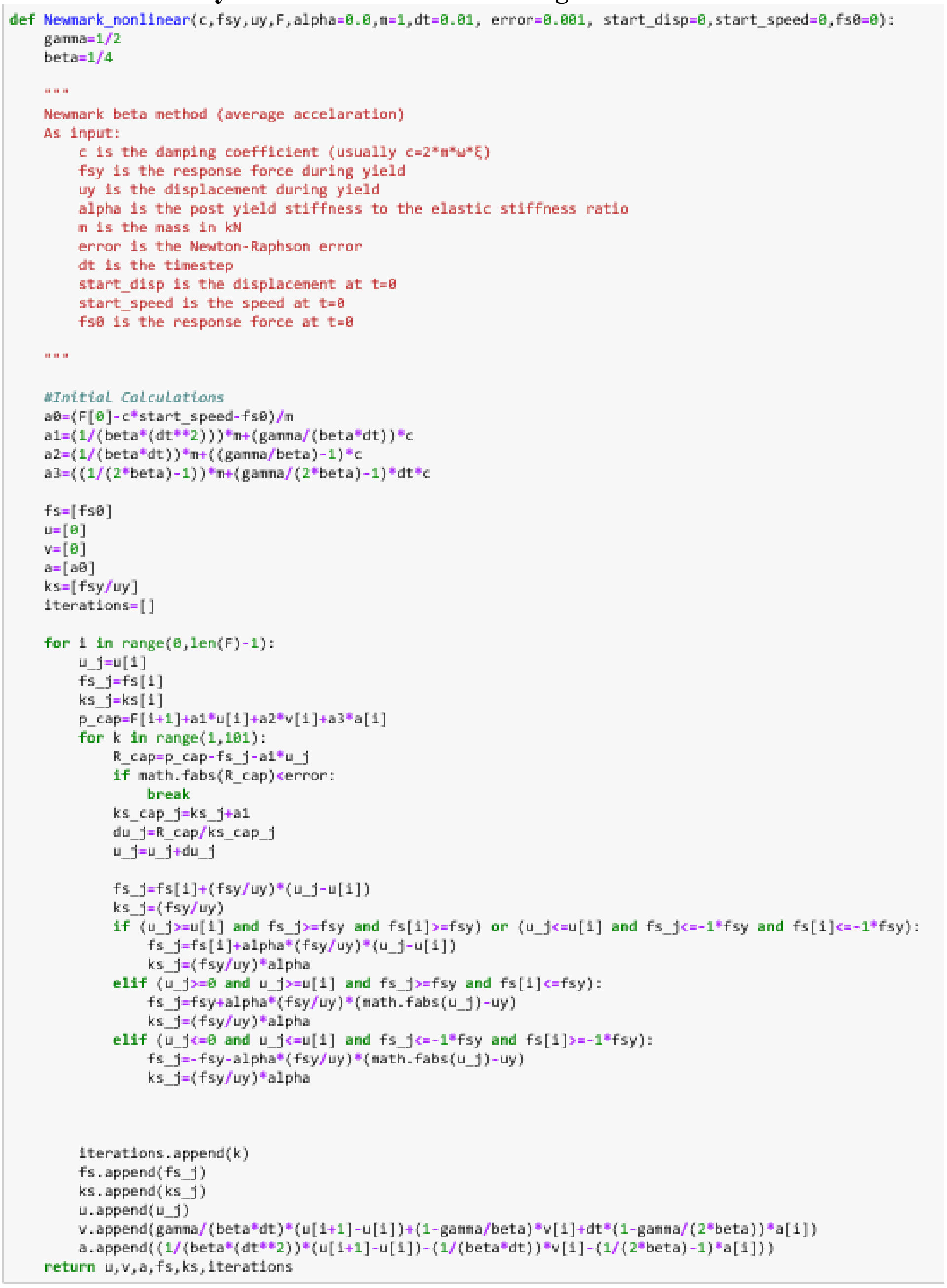
Reading ground motions from .ATC2 files (peer NGA database)

```python
#Function to read .ATC2 files downloaded from peer berkley
def load_at2(fpath):
    with open(fpath) as fp:
        for _ in range(3):
            next(fp)
        line = next(fp)
        time_step = float(line[17:25])
        accels = np.array([p for l in fp for p in l.split()]).astype(float)
    return time_step, accels
```

Linear Analysis using Newmark beta=1/4

```python
def Newmark (damp_perc,omega_system,F,dt=0.01,m=1,start_disp=0,start_speed=0):
    """
    Katsikadelis 'Dynamics of structures' page 180
    """
    #system details
    k=(omega_system**2)*m
    c=2*m*omega_system*damp_perc
    T=2*math.pi/omega_system
    start_acc = (F[0] - c*start_speed -k*start_disp)/m
    k_star = k + 2*c/dt + 4*m/(dt**2)

    #Loop
    u=[start_disp]
    v=[start_speed]
    a=[start_acc]

    u_step=start_disp
    v_step = start_speed
    a_step=start_acc

    for i in range(0,len(F)-1):
        #1
        dp= (F[i+1] -F[i])
        dp_star=dp+((4*m/dt)+2*c)*v_step+2*m*a_step

        #2
        du=dp_star/k_star

        #3
        dv= (2/dt)*du - 2*v_step
        da= (4/(dt**2))*(du-dt*v_step)-2*a_step

        #4
        u_step  = u_step + du
        v_step  = v_step + dv
        a_step  = a_step + da

        u.append(u_step)
        v.append(v_step)
        a.append(a_step)


    return u,v,a
```

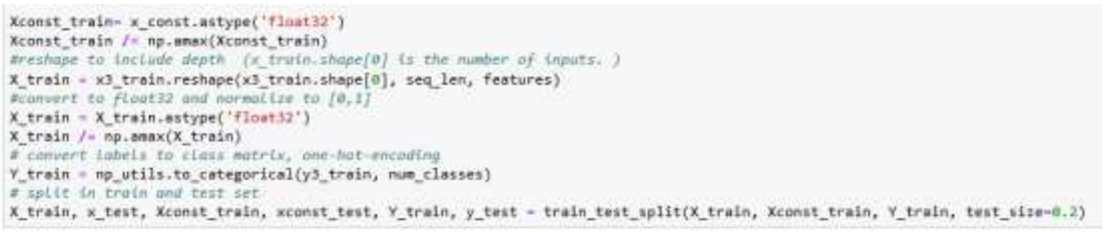
Non-Linear Analysis with Newmark's Average acceleration method



```python
def Newmark_nonlinear(c,fsy,uy,F,alpha=0.0,m=1,dt=0.01, error=0.001, start_disp=0,start_speed=0,fs0=0):
    gamma=1/2
    beta=1/4

    """
    Newmark beta method (average accelaration)
    As input:
        c is the damping coefficient (usually c=2*m*w*ξ)
        fsy is the response force during yield
        uy is the displacement during yield
        alpha is the post yield stiffness to the elastic stiffness ratio
        m is the mass in kN
        error is the Newton-Raphson error
        dt is the timestep
        start_disp is the displacement at t=0
        start_speed is the speed at t=0
        fs0 is the response force at t=0

    """

    #Initial Calculations
    a0=(F[0]-c*start_speed-fs0)/m
    a1=(1/(beta*(dt**2)))*m+(gamma/(beta*dt))*c
    a2=(1/(beta*dt))*m+((gamma/beta)-1)*c
    a3=((1/(2*beta)-1))*m+(gamma/(2*beta)-1)*dt*c

    fs=[fs0]
    u=[0]
    v=[0]
    a=[a0]
    ks=[fsy/uy]
    iterations=[]

    for i in range(0,len(F)-1):
        u_j=u[i]
        fs_j=fs[i]
        ks_j=ks[i]
        p_cap=F[i+1]+a1*u[i]+a2*v[i]+a3*a[i]
        for k in range(1,101):
            R_cap=p_cap-fs_j-a1*u_j
            if math.fabs(R_cap)<error:
                break
            ks_cap_j=ks_j+a1
            du_j=R_cap/ks_cap_j
            u_j=u_j+du_j

            fs_j=fs[i]+(fsy/uy)*(u_j-u[i])
            ks_j=(fsy/uy)
            if (u_j>=u[i] and fs_j>=fsy and fs[i]>=fsy) or (u_j<=u[i] and fs_j<=-1*fsy and fs[i]<=-1*fsy):
                fs_j=fs[i]+alpha*(fsy/uy)*(u_j-u[i])
                ks_j=(fsy/uy)*alpha
            elif (u_j>=0 and u_j>=u[i] and fs_j>=fsy and fs[i]<=fsy):
                fs_j=fsy+alpha*(fsy/uy)*(math.fabs(u_j)-uy)
                ks_j=(fsy/uy)*alpha
            elif (u_j<=0 and u_j<=u[i] and fs_j<=-1*fsy and fs[i]>=-1*fsy):
                fs_j=-fsy-alpha*(fsy/uy)*(math.fabs(u_j)-uy)
                ks_j=(fsy/uy)*alpha



        iterations.append(k)
        fs.append(fs_j)
        ks.append(ks_j)
        u.append(u_j)
        v.append(gamma/(beta*dt)*(u[i+1]-u[i])+(1-gamma/beta)*v[i]+dt*(1-gamma/(2*beta))*a[i])
        a.append((1/(beta*(dt**2))*(u[i+1]-u[i])-(1/(beta*dt))*v[i]-(1/(2*beta)-1)*a[i]))
    return u,v,a,fs,ks,iterations
```

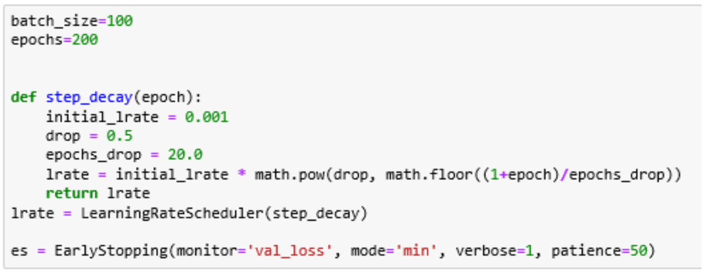
Sequential Model Example using Keras

- Reshaping data and train/test split



```
Xconst_train= x_const.astype('float32')
Xconst_train /= np.amax(Xconst_train)
#reshape to include depth  (x_train.shape[0] is the number of inputs. )
X_train = x3_train.reshape(x3_train.shape[0], seq_len, features)
#convert to float32 and normalize to [0,1]
X_train = X_train.astype('float32')
X_train /= np.amax(X_train)
# convert labels to class matrix, one-hot-encoding
Y_train = np_utils.to_categorical(y3_train, num_classes)
# split in train and test set
X_train, x_test, Xconst_train, xconst_test, Y_train, y_test = train_test_split(X_train, Xconst_train, Y_train, test_size=0.2)
```

- Learning rate function and Early stoppage



```
batch_size=100
epochs=200


def step_decay(epoch):
    initial_lrate = 0.001
    drop = 0.5
    epochs_drop = 20.0
    lrate = initial_lrate * math.pow(drop, math.floor((1+epoch)/epochs_drop))
    return lrate
lrate = LearningRateScheduler(step_decay)

es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=50)
```

- Model

```
model1 = Sequential()

model1.add(Conv1D(32, 80, activation='relu', input_shape=(seq_len, features)))
model1.add(MaxPool1D(pool_size=8))
model1.add(Dropout(0.5))

model1.add(Conv1D(32, 80, activation='relu'))
model1.add(MaxPool1D(pool_size=8))
model1.add(Dropout(0.5))

model1.add(Flatten())

model1.add(Dense(128, activation='relu'))

model1.add(Dense(64, activation='relu'))

model1.add(Dense(num_classes, activation='softmax'))

model1.compile(loss='categorical_crossentropy',  optimizer=optimizers.Adam(), metrics=['accuracy'])

model1.summary()
#model1.save('lenet.h5')
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_1 (Conv1D)            (None, 7921, 32)          10272
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 990, 32)           0
_________________________________________________________________
dropout_1 (Dropout)          (None, 990, 32)           0
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 911, 32)           81952
_________________________________________________________________
max_pooling1d_2 (MaxPooling1 (None, 113, 32)           0
_________________________________________________________________
dropout_2 (Dropout)          (None, 113, 32)           0
_________________________________________________________________
flatten_1 (Flatten)          (None, 3616)              0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               462976
_________________________________________________________________
dense_2 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_3 (Dense)              (None, 5)                 325
=================================================================
Total params: 563,781
Trainable params: 563,781
Non-trainable params: 0
_________________________________________________________________
```

- Training the model

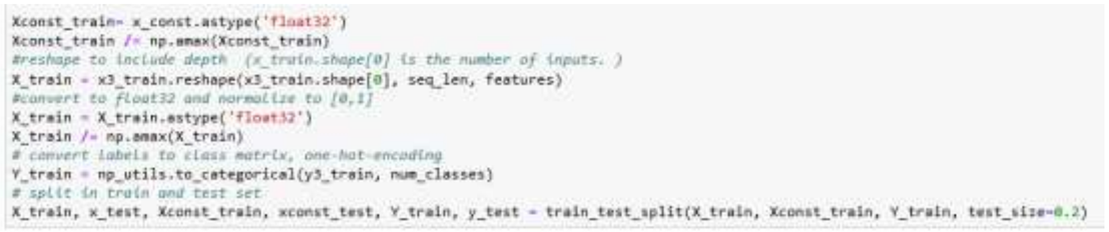
```
history1=model1.fit(X_train, Y_train, epochs=epochs, batch_size=batch_size,shuffle=True,verbose=1,
                   validation_data=(x_test, y_test), callbacks=[es,lrate])
```

- Learning Curves

```
# Loss Curves
plt.figure(figsize=[8,6])
plt.plot(history.history['loss'],'r',linewidth=1.0)
plt.plot(history.history['val_loss'],'b',linewidth=1.0)
plt.legend(['Training loss', 'Validation Loss'],fontsize=18)
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Loss',fontsize=16)
plt.title('Loss Curves',fontsize=16)

# Accuracy Curves
plt.figure(figsize=[8,6])
plt.plot(history.history['acc'],'r',linewidth=1.0)
plt.plot(history.history['val_acc'],'b',linewidth=1.0)
plt.legend(['Training Accuracy', 'Validation Accuracy'],fontsize=18)
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Accuracy',fontsize=16)
plt.title('Accuracy Curves',fontsize=16)
```

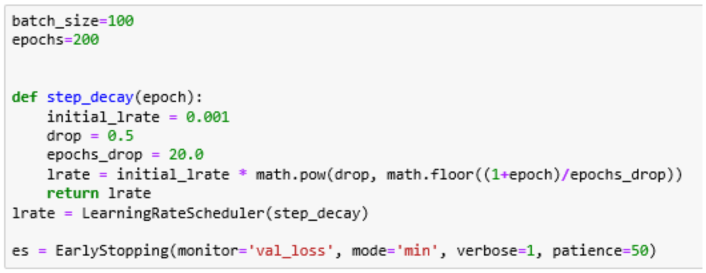
Functional API Model Example using Keras

- Reshaping data and train/test split



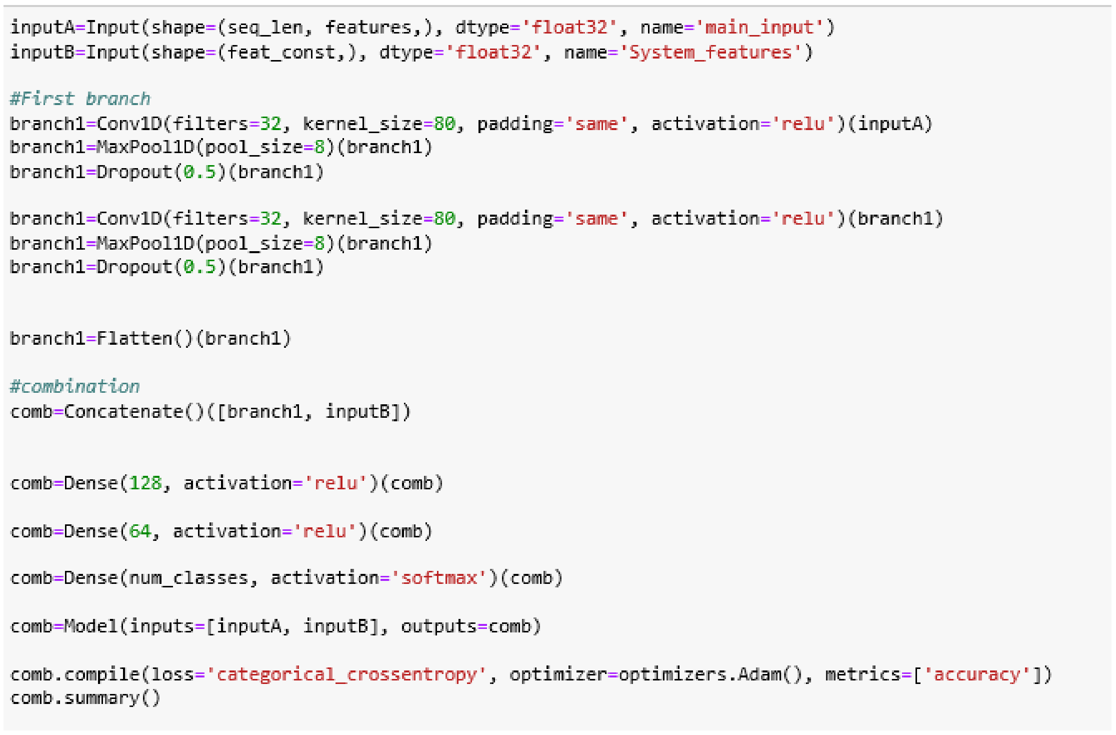
```
Xconst_train= x_const.astype('float32')
Xconst_train /= np.amax(Xconst_train)
#reshape to include depth  (x_train.shape[0] is the number of inputs. )
X_train = x3_train.reshape(x3_train.shape[0], seq_len, features)
#convert to float32 and normalize to [0,1]
X_train = X_train.astype('float32')
X_train /= np.amax(X_train)
# convert labels to class matrix, one-hot-encoding
Y_train = np_utils.to_categorical(y3_train, num_classes)
# split in train and test set
X_train, x_test, Xconst_train, xconst_test, Y_train, y_test = train_test_split(X_train, Xconst_train, Y_train, test_size=0.2)
```

- Learning rate function and Early stoppage



```
batch_size=100
epochs=200


def step_decay(epoch):
    initial_lrate = 0.001
    drop = 0.5
    epochs_drop = 20.0
    lrate = initial_lrate * math.pow(drop, math.floor((1+epoch)/epochs_drop))
    return lrate
lrate = LearningRateScheduler(step_decay)

es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=50)
```

- Model



```
inputA=Input(shape=(seq_len, features,), dtype='float32', name='main_input')
inputB=Input(shape=(feat_const,), dtype='float32', name='System_features')

#First branch
branch1=Conv1D(filters=32, kernel_size=80, padding='same', activation='relu')(inputA)
branch1=MaxPool1D(pool_size=8)(branch1)
branch1=Dropout(0.5)(branch1)

branch1=Conv1D(filters=32, kernel_size=80, padding='same', activation='relu')(branch1)
branch1=MaxPool1D(pool_size=8)(branch1)
branch1=Dropout(0.5)(branch1)


branch1=Flatten()(branch1)

#combination
comb=Concatenate()([branch1, inputB])


comb=Dense(128, activation='relu')(comb)

comb=Dense(64, activation='relu')(comb)

comb=Dense(num_classes, activation='softmax')(comb)

comb=Model(inputs=[inputA, inputB], outputs=comb)

comb.compile(loss='categorical_crossentropy', optimizer=optimizers.Adam(), metrics=['accuracy'])
comb.summary()
```

```
____________________________________________________________________________________________________
Layer (type)                     Output Shape          Param #     Connected to
====================================================================================================
main_input (InputLayer)          (None, 8000, 4)       0
____________________________________________________________________________________________________
conv1d_1 (Conv1D)                (None, 8000, 32)      10272       main_input[0][0]
____________________________________________________________________________________________________
max_pooling1d_1 (MaxPooling1D)   (None, 1000, 32)      0           conv1d_1[0][0]
____________________________________________________________________________________________________
dropout_1 (Dropout)              (None, 1000, 32)      0           max_pooling1d_1[0][0]
____________________________________________________________________________________________________
conv1d_2 (Conv1D)                (None, 1000, 32)      81952       dropout_1[0][0]
____________________________________________________________________________________________________
max_pooling1d_2 (MaxPooling1D)   (None, 125, 32)       0           conv1d_2[0][0]
____________________________________________________________________________________________________
dropout_2 (Dropout)              (None, 125, 32)       0           max_pooling1d_2[0][0]
____________________________________________________________________________________________________
flatten_1 (Flatten)              (None, 4000)          0           dropout_2[0][0]
____________________________________________________________________________________________________
System_features (InputLayer)     (None, 2)             0
____________________________________________________________________________________________________
concatenate_1 (Concatenate)      (None, 4002)          0           flatten_1[0][0]
                                                                   System_features[0][0]
____________________________________________________________________________________________________
dense_1 (Dense)                  (None, 128)           512384      concatenate_1[0][0]
____________________________________________________________________________________________________
dense_2 (Dense)                  (None, 64)            8256        dense_1[0][0]
____________________________________________________________________________________________________
dense_3 (Dense)                  (None, 3)             195         dense_2[0][0]
====================================================================================================
Total params: 613,059
Trainable params: 613,059
Non-trainable params: 0
____________________________________________________________________________________________________
```

- Training the model

```python
history=comb.fit([X_train, Xconst_train], Y_train, epochs=epochs, batch_size=batch_size,shuffle=True,verbose=1,
                 validation_data=([x_test,xconst_test], y_test),  callbacks=[es,lrate])
```

- Learning Curves

```python
# Loss Curves
plt.figure(figsize=[8,6])
plt.plot(history.history['loss'],'r',linewidth=1.0)
plt.plot(history.history['val_loss'],'b',linewidth=1.0)
plt.legend(['Training loss', 'Validation Loss'],fontsize=18)
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Loss',fontsize=16)
plt.title('Loss Curves',fontsize=16)

# Accuracy Curves
plt.figure(figsize=[8,6])
plt.plot(history.history['acc'],'r',linewidth=1.0)
plt.plot(history.history['val_acc'],'b',linewidth=1.0)
plt.legend(['Training Accuracy', 'Validation Accuracy'],fontsize=18)
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Accuracy',fontsize=16)
plt.title('Accuracy Curves',fontsize=16)
```