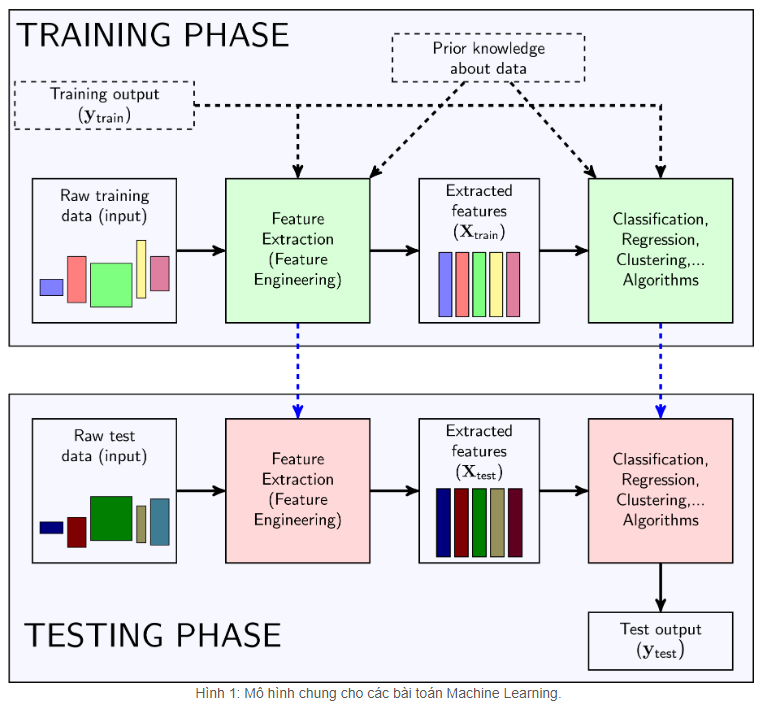
Khi làm việc với các bài toán Machine Learning thực tế, nhìn chung chúng ta chỉ có được dữ liệu thô (raw) chưa qua chỉnh sửa, chọn lọc. Chúng ta cần phải tìm một phép biến đổi để loại ra những dữ liệu nhiễu (noise), và để đưa dữ liệu thô với số chiều khác nhau về cùng một chuẩn (cùng là các vector hoặc ma trận). Dữ liệu chuẩn mới này phải đảm bảo giữ được những thông tin đặc trưng (features) cho dữ liệu thô ban đầu. Không những thế, tùy vào từng bài toán, ta cần thiết kế những phép biến đổi để có những features phù hợp. Quá trình quan trọng này được gọi là Feature Extraction, hoặc Feature Engineering, một số tài liệu tiếng Việt gọi nó là trích chọn đặc trưng.

Mô hình chung cho các bài toán Machine Learning



TRAINING PHASE

Feature Extractor

**ĐẦU RA**

Mục đích của Feature Engineering là tạo ra một Feature Extractor biến dữ liệu thô ban đầu thành dữ liệu phù hợp với từng mục đích khác nhau.

ĐẦU VÀO

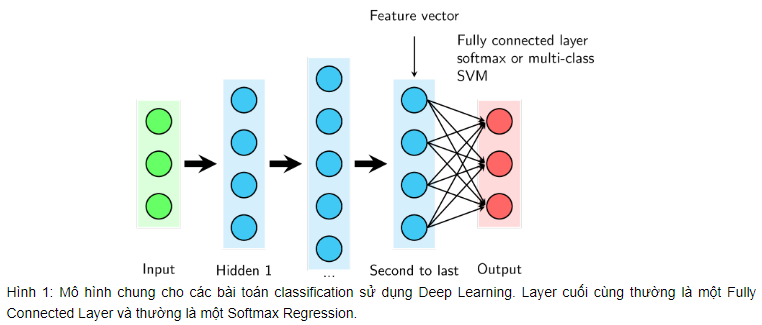
raw training input. Raw input là tất cả các thông tin ta biết về dữ liệu.

Feature selection

Giả sử rằng các điểm dữ liệu có số features khác nhau (do kích thước dữ liệu khác nhau hay do một số feature mà điểm dữ liệu này có nhưng điểm dữ liệu kia lại không thu thập được), và số lượng features là cực lớn. Chúng ta cần *chọn* ra một số lượng nhỏ hơn các feature phù hợp với bài toán.

Transfer Learning cho bài toán phân loại ảnh

Các phương pháp Feature Engineering nêu trên thường được gọi là các *hand-crafted features* (feature được tạo thủ công) vì nó chủ yếu dựa trên các quan sát về đặc tính riêng của ảnh. Các phương pháp này cho kết quả khá ấn tượng trong một số trường hợp. Tuy nhiên, chúng vẫn còn nhiều hạn chế vì quá trình tìm ra các features và các classifier phù hợp vẫn là riêng biệt.



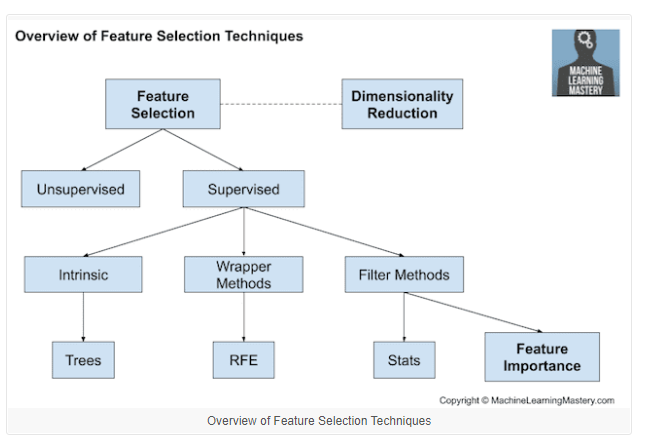
output ở layer gần cuối cùng (second to last layer) có thể được coi là feature vectors và Softmax Regression chính là Classifier được sử dụng.

Như đã đề cập, toàn bộ các layer trừ output layer có thể được coi là một bộ Feature Extractor. Dựa trên nhận xét rằng các bức ảnh đều có những đặc tính giống nhau nào đó, với cơ sở dữ liệu khác, ta cũng có thể sử dụng phần Feature Extractor này để tạo ra các feature vectors. Sau đó, ta thay output layer cũng bằng một Softmax Regression (hoặc multi-class SVM) nhưng với số lượng units bằng với số lượng class ở bộ cơ sở dữ liệu mới. Ta chỉ cần train layer cuối cùng này. Kinh nghiệm thực tế của tôi cho thấy, việc làm này đã tăng kết quả phân lớp lên rất nhiều so với việc sử dụng các hand-crafted features.

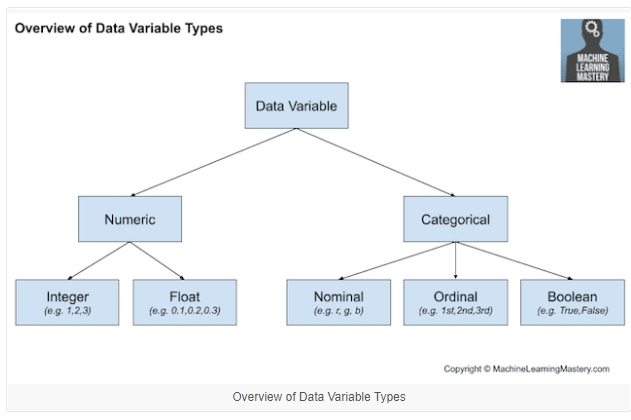
Cách làm như trên được gọi là **ConvNet as fixed feature extractor**, tức ta sử dụng trực tiếp vector ở second to last layer làm feature vector. Nếu tiếp tục tinh chỉnh (Fine-tuning) một chút nữa, kết quả sẽ có thể tốt hơn.

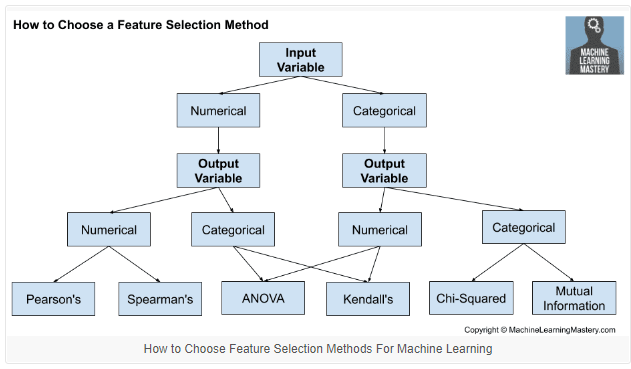
<https://cs231n.github.io/transfer-learning/>

https://machinelearningmastery.com/feature-selection-with-real-and-categorical-data/

* **Feature Selection**: Select a subset of input features from the dataset.
  + - **Unsupervised**: Do not use the target variable (e.g. remove redundant variables).
      * Correlation
    - **Supervised**: Use the target variable (e.g. remove irrelevant variables).
      * **Wrapper**: Search for well-performing subsets of features.
        + RFE
      * **Filter**: Select subsets of features based on their relationship with the target.
        + Statistical Methods
        + Feature Importance Methods
      * **Intrinsic**: Algorithms that perform automatic feature selection during training.
        + Decision Trees
* **Dimensionality Reduction**: Project input data into a lower-dimensional feature space

## Statistics for Filter-Based Feature Selection Methods





### Correlation Statistics

The scikit-learn library provides an implementation of most of the useful statistical measures.

For example:

* Pearson’s Correlation Coefficient: [f\_regression()](https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.f_regression.html)
* ANOVA: [f\_classif()](https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.f_classif.html)
* Chi-Squared: [chi2()](https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.chi2.html)
* Mutual Information: [mutual\_info\_classif()](https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.mutual_info_classif.html) and [mutual\_info\_regression()](https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.mutual_info_regression.html)

Also, the SciPy library provides an implementation of many more statistics, such as Kendall’s tau ([kendalltau](https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.kendalltau.html)) and Spearman’s rank correlation ([spearmanr](https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.spearmanr.html)).

### Selection Method

The scikit-learn library also provides many different filtering methods once statistics have been calculated for each input variable with the target.

Two of the more popular methods include:

* Select the top k variables: [SelectKBest](https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html)
* Select the top percentile variables: [SelectPercentile](https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectPercentile.html)

I often use SelectKBest myself.

https://machinelearningmastery.com/feature-selection-with-categorical-data/

We can use the [OrdinalEncoder() from scikit-learn](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OrdinalEncoder.html) to encode each variable to integers. This is a flexible class and does allow the order of the categories to be specified as arguments if any such order is known.

* Dùng như LabelEncoder nhưng cho multiple inputs

<https://machinelearningmastery.com/encoder-decoder-models-text-summarization-keras/>

## Encoder-Decoder Architecture

The Encoder-Decoder architecture is a way of organizing recurrent neural networks for sequence prediction problems that have a variable number of inputs, outputs, or both inputs and outputs.

The architecture involves two components: an encoder and a decoder.

* **Encoder**: The encoder reads the entire input sequence and encodes it into an internal representation, often a fixed-length vector called the context vector.
* **Decoder**: The decoder reads the encoded input sequence from the encoder and generates the output sequence.

Both the encoder and the decoder submodels are trained jointly, meaning at the same time.

The entire encoded input is used as context for generating each step in the output. Although this works, the fixed-length encoding of the input limits the length of output sequences that can be generated.

https://machinelearningmastery.com/encoder-decoder-long-short-term-memory-networks/

The Encoder-Decoder LSTM is a recurrent neural network designed to address sequence-to-sequence problems, sometimes called seq2seq.

Sequence-to-sequence prediction problems are challenging because the number of items in the input and output sequences can vary.

## Sequence-to-Sequence Prediction Problems

Sequence prediction often involves forecasting the next value in a real valued sequence or outputting a class label for an input sequence.

This is often framed as a sequence of one input time step to one output time step (e.g. one-to-one) or multiple input time steps to one output time step (many-to-one) type sequence prediction problem.

There is a more challenging type of sequence prediction problem that takes a sequence as input and requires a sequence prediction as output. These are called sequence-to-sequence prediction problems, or seq2seq for short.

One modeling concern that makes these problems challenging is that the length of the input and output sequences may vary. Given that there are multiple input time steps and multiple output time steps, this form of problem is referred to as many-to-many type sequence prediction problem.

## Encoder-Decoder LSTM Architecture

One approach to seq2seq prediction problems that has proven very effective is called the Encoder-Decoder LSTM.

This architecture is comprised of two models: one for reading the input sequence and encoding it into a fixed-length vector, and a second for decoding the fixed-length vector and outputting the predicted sequence. The use of the models in concert gives the architecture its name of Encoder-Decoder LSTM designed specifically for seq2seq problems.

The Encoder-Decoder LSTM was developed for natural language processing problems where it demonstrated state-of-the-art performance, specifically in the area of text translation called statistical machine translation.

The innovation of this architecture is the use of a fixed-sized internal representation in the heart of the model that input sequences are read to and output sequences are read from. For this reason, the method may be referred to as sequence embedding.

This approach has also been used with image inputs where a Convolutional Neural Network is used as a feature extractor on input images, which is then read by a decoder LSTM.

An extension of the Encoder-Decoder architecture is to provide a more expressive form of the encoded input sequence and allow the decoder to learn where to pay attention to the encoded input when generating each step of the output sequence.

This extension of the architecture is called attention.

<https://machinelearningmastery.com/attention-long-short-term-memory-recurrent-neural-networks/>

<https://machinelearningmastery.com/timedistributed-layer-for-long-short-term-memory-networks-in-python/>

One reason for this difficulty in Keras is the use of the TimeDistributed wrapper layer and the need for some LSTM layers to return sequences rather than single values.

## TimeDistributed Layer

An added complication is the [TimeDistributed](https://keras.io/layers/wrappers/" \l "timedistributed) Layer (and the former TimeDistributedDense layer) that is cryptically described as a layer wrapper:

*TimeDistributedDense applies a same Dense (fully-connected) operation to every timestep of a 3D tensor.*

The input for LSTMs must be three dimensional.

There are two key points to remember when using the TimeDistributed wrapper layer:

* **The input must be (at least) 3D**. This often means that you will need to configure your last LSTM layer prior to your TimeDistributed wrapped Dense layer to return sequences (e.g. set the “return\_sequences” argument to “True”).
* **The output will be 3D**. This means that if your TimeDistributed wrapped Dense layer is your output layer and you are predicting a sequence, you will need to resize your y array into a 3D vector.

# [**Is it possible to give variable sized images as input to a convolutional neural network?**](https://stats.stackexchange.com/questions/388859/is-it-possible-to-give-variable-sized-images-as-input-to-a-convolutional-neural)

<https://stats.stackexchange.com/questions/388859/is-it-possible-to-give-variable-sized-images-as-input-to-a-convolutional-neural>

There are a number of ways to do it. Most of these have already been covered in a number of posts over StackOverflow, Quora and other content websites.

To summarize, most of the techniques listed can be grouped into two classes of solutions, namely,

1. **Transformations**
2. Inherent **Network Property**

In transformations, one can look up techniques such as

* **Resize**, which is the simplest of all the techniques mentioned
* **Crop**, which can be done as a sliding window or one-time crop with information loss

One can also look into networks that have inherent property to be immune to the size of the input by the virtue of layer behaviour which builds up the network. Examples of this can be found in terms of,

* **Fully convolutional networks (FCN)**, which have no limitations on the input size at all because once the kernel and step sizes are described, the convolution at each layer can generate appropriate dimension outputs according to the corresponding inputs.
* **Spatial Pyramid Pooling (SPP)**, FCNs do not have a fully connected dense layer and hence are agnostic to the image size, but say if one wanted to use dense layer without considering input transformations, then there is a interesting [paper](https://arxiv.org/abs/1406.4729) that explains the layer in a deep learning network.

<https://arxiv.org/abs/1406.4729>

The convolutional layers and pooling layers themselves are independent of the input dimensions. However, the output of the convolutional layers will have different spatial sizes for differently sized images, and this will cause an issue if we have a fully connected layer afterwards (since our fully connected layer requires a fixed size input). There are several solutions to this:

**1. Global Pooling:** Avoid fully connected layers at the end of the convolutional layers, and instead use pooling (such as Global Average Pooling) to reduce your feature maps from a shape of (N,H,W,C) (before global pool) to shape (N,1,1,C) (after global pool), where:  
  
N = Number of minibatch samples  
H = Spatial height of feature map  
W = Spatial width of feature map  
C = Number of feature maps (channels)  
  
As can be seen, the output dimensionality (N\*C) is now independent of the spatial size (H,W) of the feature maps. In case of classification, you can then proceed to use a fully connected layer on top to get the logits for your classes.  
  
**2. Variable sized pooling:** Use variable sized pooling regions to get the same feature map size for different input sizes.  
  
**3. Crop/Resize/Pad input images:** You can try to rescale/crop/pad your input images to all have the same shape.

In the context of transfer learning, you might want to use differently sized inputs than the original inputs that the model was trained with. Here are some options for doing so:  
  
**4. Create new fully connected layers:** You can ditch the original fully connected layers completely and initialize a new fully connected layer with the dimensionality that you need, and train it from scratch.  
  
**5. Treat the fully connected layer as a convolution:** Normally, we reshape the feature maps from (N,H,W,C) to (N,H\*W\*C) before feeding it to the fully connected layer. But you can also treat the fully connected layer as a convolution with a receptive field of (H,W). Then, you can just convolve this kernel with your feature maps regardless of their size (use zero padding if needed)

<https://ai.stackexchange.com/questions/2008/how-can-neural-networks-deal-with-varying-input-sizes>

<https://datascience.stackexchange.com/questions/14664/neural-network-with-flexible-number-of-inputs>

Yes this is possible by treating the audio as a sequence into a [Recurrent Neural Network (RNN)](https://deeplearning4j.konduit.ai/getting-started/tutorials/recurrent-networks). You can train a RNN against a target that is correct at the end of a sequence, or even to predict another sequence offset from the input.

Do note however that there is [a bit to learn about options that go into the construction and training of a RNN](https://iamtrask.github.io/2015/11/15/anyone-can-code-lstm/), that you will not already have studied whilst looking at simpler layered feed-forward networks. Modern RNNs make use of layer designs which include memory gates - the two most popular architectures are LSTM and GRU, and these add more trainable parameters into each layer as the memory gates need to learn weights in addition to the weights between and within the layer.

RNNs are used extensively to predict from audio sequences that have already been processed in MFCC or similar feature sets, because they can handle sequenced data as input and/or output, and this is a desirable feature when dealing with variable length data such as [spoken word](https://arxiv.org/abs/1402.1128), music etc.

Some other things worth noting:

* RNNs can work well for *sequences* of data that are variable length, and where there is a well-defined dimension over which the sequences evolve. But they are less well adapted for variable-sized sets of features where there is no clear order or sequence.
* RNNs can get state-of-the-art results for signal processing, NLP and related tasks, but only when there is a very large amount of training data. Other, simpler, models can work just as well or better if there is less data.
* For the specific problem of generating MFCCs from raw audio samples: Whilst it should be possible to create a RNN that predicts MFCC features from raw audio, this might take some effort and experimentation to get right, and could take a lot of processing power to make an RNN powerful enough to cope with very long sequences at normal audio sample rates. Whilst creating MFCC from raw audio using the standard approach starting with FFT will be a lot simpler, and is guaranteed to be accurate.