Introduction: Beyond the Feature dimension

Adding ”shape” to an example

Thus far, the data examples we have been using are vectors

- the only dimension is the "feature" dimension
- for example, the names of the features in the feature dimension are `Price, Volume, Open, Close`

The diagram shows (our typical, up to now: 0 non-feature dimensional) feature vector $\mathbf{x}$ matched against pattern $\mathbf{k}$

- where the feature dimension is length 3

Zero non-feature dimensions, length 3 feature dimension

In this module, we extend "pattern matching" to includes examples that have "shape"

- an arrangement of *elements*, each element being a vector of features

The arrangement of elements will be described by

- dimensions beyond the feature dimension

For example

- a timeseries
    - elements arranged as a *sequence* via a single non-feature dimension called "time"
    - each element is a vector with features `Price, Volume, Open, Close`
- a pixel grid
    - elements arranged in two dimensional space with non-feature dimensions called "row" and "column"
    - each element has the features `Red, Green, Blue`

Consider an example with

- one non-feature dimension of length 2 (horizontal axis)
$$d_1 = 2$$
- three features
$$n = 3$$

One non-feature dimension of length 2, feature dimension of length 3

Spatial

The example $\mathbf{x}$ has two *elements*

- an *element* is a vector with *no* non-feature dimensions
- labeled $\mathbf{x}_{[0]}$ and $\mathbf{x}_{[1]}$
- each of length n

This is a one-dimensional (counting only non-feature dimensions) example.

You might see an example like this when dealing with data from Equity pricing

- features: `Close Price, Open Price, Volume`
- the elements might correspond to
    - different equities
    - different dates

We can generalize to more than a single non-feature dimension

For example

- an $(H \times W)$ image has two non-feature dimensions with lengths
$$d_1 = H, d_2 = W$$

with $H * W$ elements.

In general

- if there are N non-feature dimensions
    - where the length of the i^{th} dimensions is denoted d_i
- we can index an element by a vector of length N in
$$[1:d_1] \times [1:d_2] \times \ldots [1:d_N]$$
- an index identifies a specific *location* in the non-feature dimensions

There are $\prod_{i=1}^N d_i$ elements in the example

- one at each location
- where the location is specified by its index in the non-feature dimensions

Pattern matching examples with shape

How does pattern matching work in the presence of non-feature dimensions ?

Consider an example $\mathbf{x}$ with N non-feature dimensions of lengths

$$d_1, \ldots, d_N$$

We will define a pattern $\mathbf{k}$ to have *identical* shape as the example

- same number N of non-feature dimensions
- same lengths of these dimensions
    - we will subsequently allow the lengths to be shorter
- same number of features
- define elements of the pattern similar to elements of the example
    - there are $\prod_{i=1}^N d_i$ elements in the pattern

Remember

- The shape of a "full" pattern is the same as the shape of an example

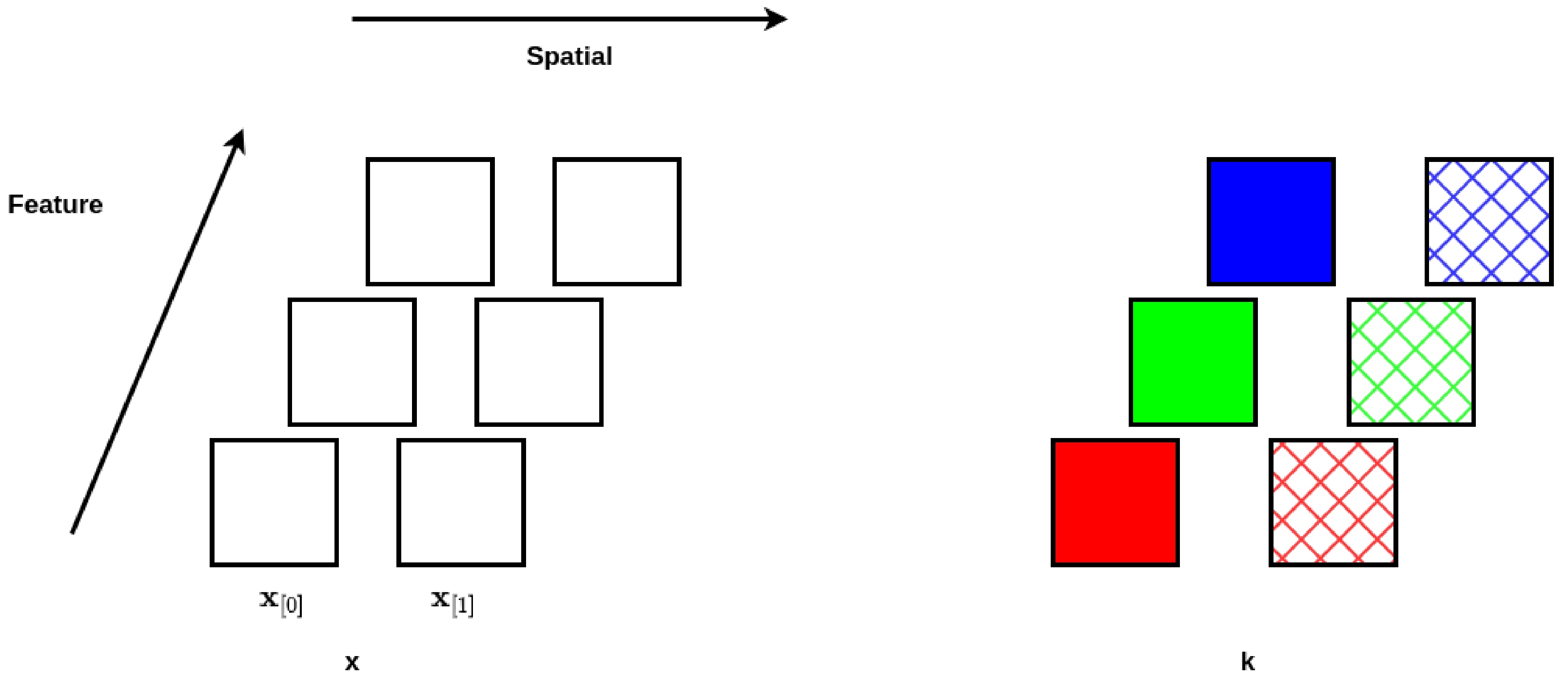
Terminology

In the literature of Convolutions, some familiar concepts are described with different words

- A pattern is also referred to as a *kernel*
- The feature dimensions is also referred to as the *channel* dimension

Here is an example $\mathbf{x}$ along with pattern $\mathbf{k}$

One non-feature dimension of length 2, feature dimension of length 3



We now generalize the dot product to accommodate non-feature dimensions

$$\mathbf{x} \cdot \mathbf{k} = \sum_{\text{idx} \in [1:d_1] \times [1:d_2] \times \ldots [1:d_N]} \mathbf{x}_{\text{idx}} \cdot \mathbf{k}_{\text{idx}}$$

That is

- we perform the dot product of feature-only vectors
- of elements in $\mathbf{x}$ and $\mathbf{k}$ with identical indices
- and sum them up

$$\mathbf{x}_{[0]} \cdot \mathbf{k}_{[0]} + \mathbf{x}_{[1]} \cdot \mathbf{k}_{[1]}$$

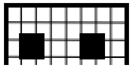
So the *scalar* result may be interpreted

- the dot product of each element matches similarity of corresponding elements
- the sum creates a kind of "average similarity" across the elements

Let's visualize the generalized dot product with a more familiar example

- recognizing a smiley face in a 2D image

Two non-feature dimensions, each of length 8, feature dimension of length `
One pattern



$\mathbf{y}_{(l-1)}$

In the above diagram

- the example has non-spatial dimensions $d_1 = d_2 = 8$
- one feature: $n = 1$
- there is one pattern
    - non-feature and feature dimensions identical to example
    - a "full" pattern
- the number of features of the output
    - equals the number of patterns
    - one output feature

Multiple output features: matching against multiple patterns

The above matches an example $\mathbf{x}$ with a single pattern

- to create a single output feature

We can create a *second* output feature by adding a second pattern

- similar to how a Fully Connected layer creates multiple features via multiple patterns
- resulting in an output vector consisting of 2 features
    - a no non-feature dimensions

In general, we can add many patterns

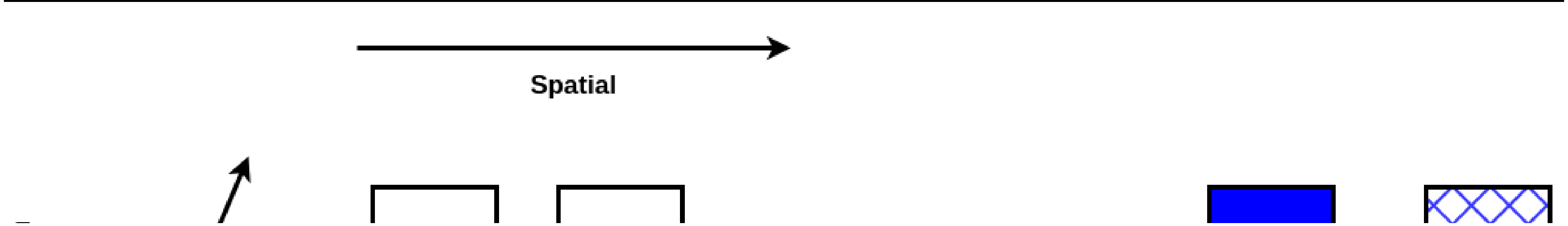
The diagram shows a

- 1 non-feature dimension (of length 2) vector $\mathbf{x}$ $d_1 = 2$
  - with feature dimension length 3

$$n = 3$$

- matched against 2 patterns $\mathbf{k}_0, \mathbf{k}_1$
  - pattern $\mathbf{k}_i$ has elements denoted $\mathbf{k}_{i,[0]}$ and $\mathbf{k}_{i,[1]}$
- resulting in an output with 2 features
  - the first measuring the intensity of the match with the first pattern
  - the second measuring the intensity of the match with the second pattern

One non-feature dimension of length 2, feature dimension of length 3
Two patterns


Spatial

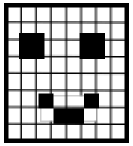
Visualizing the dot product with our "smiley face" example once more

- second pattern is a weaker match with the example than first pattern
- two patterns means output has two features

Remember

The number of output features is equal to the number of kernels/patterns

Two non-feature dimensions, each of length 8, feature dimension of length 1
Two patterns



$$\mathbf{y}_{(l-1)}$$

$$\underbrace{8 \times 8}_{\textbf{Spatial}} \times \underbrace{1}_{\textbf{Channel}}$$

Patterns smaller than examples

Thus far, the non-feature dimensions of the example and pattern

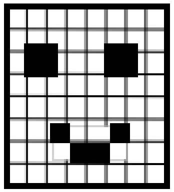
- are identical in number N
- and lengths $d_1, \ldots d_N$

That is

- we seek to match a pattern against the entire non-feature dimensions of the input.

Consider the following pattern which is of identical dimension to the input

Pattern spanning entire non-feature dimensions, single feature



$$\mathbf{y}_{(l-1)}$$

$$\underbrace{8 \times 8}_{\textbf{Spatial}} \times \underbrace{1}_{\textbf{Channel}}$$

In fact: this pattern is identical to the input and matches it perfectly !

- the generalized dot product of the input and the pattern results in a high activation

But what about examples similar to this one but

- shifted right/left or up/down
    - we week "translational invariance"
- a smaller smile
- different distance between the eyes

The pattern would not be as good a match

- lower activation
- even though the "meaning" of the similar input is the same as the original: "smiling face"

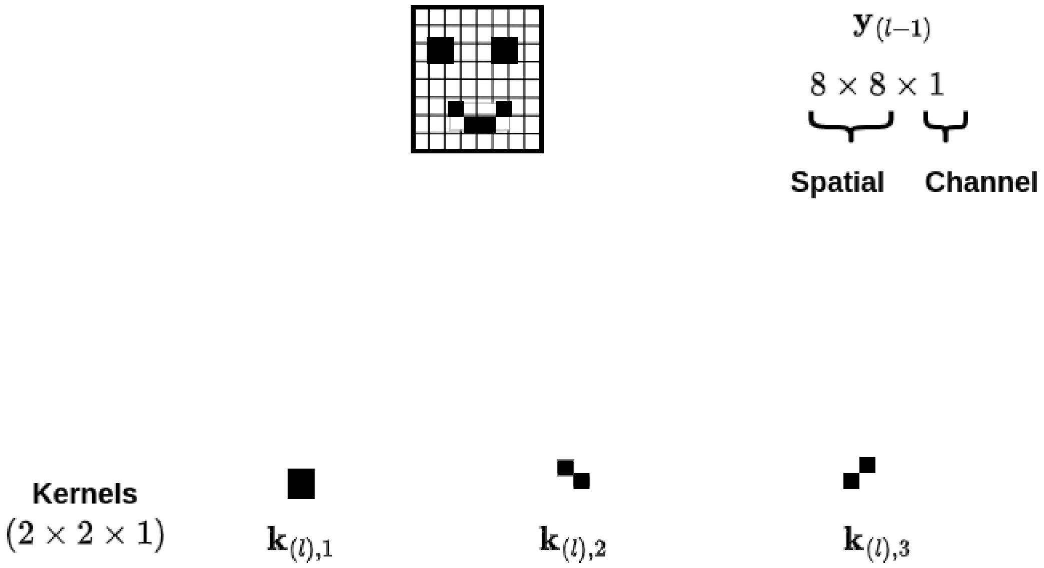
It might be useful to be able to match

- smaller patterns
- that occur *somewhere* in the example
- rather than a pattern that matches the entire example

Consider the following smaller (2×2) patterns

- one matching an eye
- one matching the left corner of a mouth
- one matching the right corner of a mouth

Convolution: 1 input feature to 3 output features



Does the first pattern (the eye) occur *somewhere* in $\mathbf{x}$?

We define an operation to answer that questions.

Convolution: visual explanation

It will be easier to describe the operation via a picture

- and follow up with a more precise formulation

We match an "eye" pattern

- against every sub-region of the example
- of identical size to the pattern

What does it mean to match the pattern against every "sub-region" ?

Imagine placing the pattern to overlap the upper left corner (one region)

- match the pattern against this sub-region
- this measures the intensity of the match at this particular sub-region

Now move this pattern (e.g., one pixel right/left or up/down)

- this defines another sub-region against which the pattern is matched
- the match measures the intensity of the match at this particular sub-region

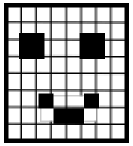
Repeat this matching process against every sub-region.

The result

- has shape (in the non-feature dimensions) that is the same as the example's shape (in the non-feature dimensions)
- whose value is an intensity

A picture will make this more concrete.

Convolution: 1 input feature to 1 output feature

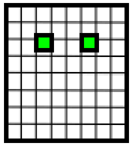


$$\mathbf{y}_{(l-1)}$$

$$\underbrace{8 \times 8}_{\textbf{Spatial}} \times \underbrace{1}_{\textbf{Channel}}$$

Kernels
$(2 \times 2 \times 1)$

$\mathbf{k}_{(l),1}$



$\mathbf{y}_{(l),1}$

$$\mathbf{y}_{(l)}$$

$$\underbrace{8 \times 8}_{\textbf{Spatial}} \times \underbrace{1}_{\textbf{Channel}}$$

In words:

- the "small" pattern $\mathbf{k}$ (less than full non-feature dimension size of the example)
- is matched against each "sub-region" of $\mathbf{x}$
    - where a sub-region has non-feature dimensions that are the same as the kernel
    - is centered at one index idx in the set of element indices of $\mathbf{x}$
- producing a scalar value
    - indicating the intensity of the match of the small pattern with the part centered at idx
- the output
    - has the same non-feature dimensions as the input example
    - has one feature
        - the match intensity

The output is called a *feature map*

- same non-feature dimension "shape" as example
    - N non-feature dimensions of lengths $d_1, \ldots d_N$
- maps the intensity of the match of the pattern
- when the pattern is centered at each index in the set of indices of the example

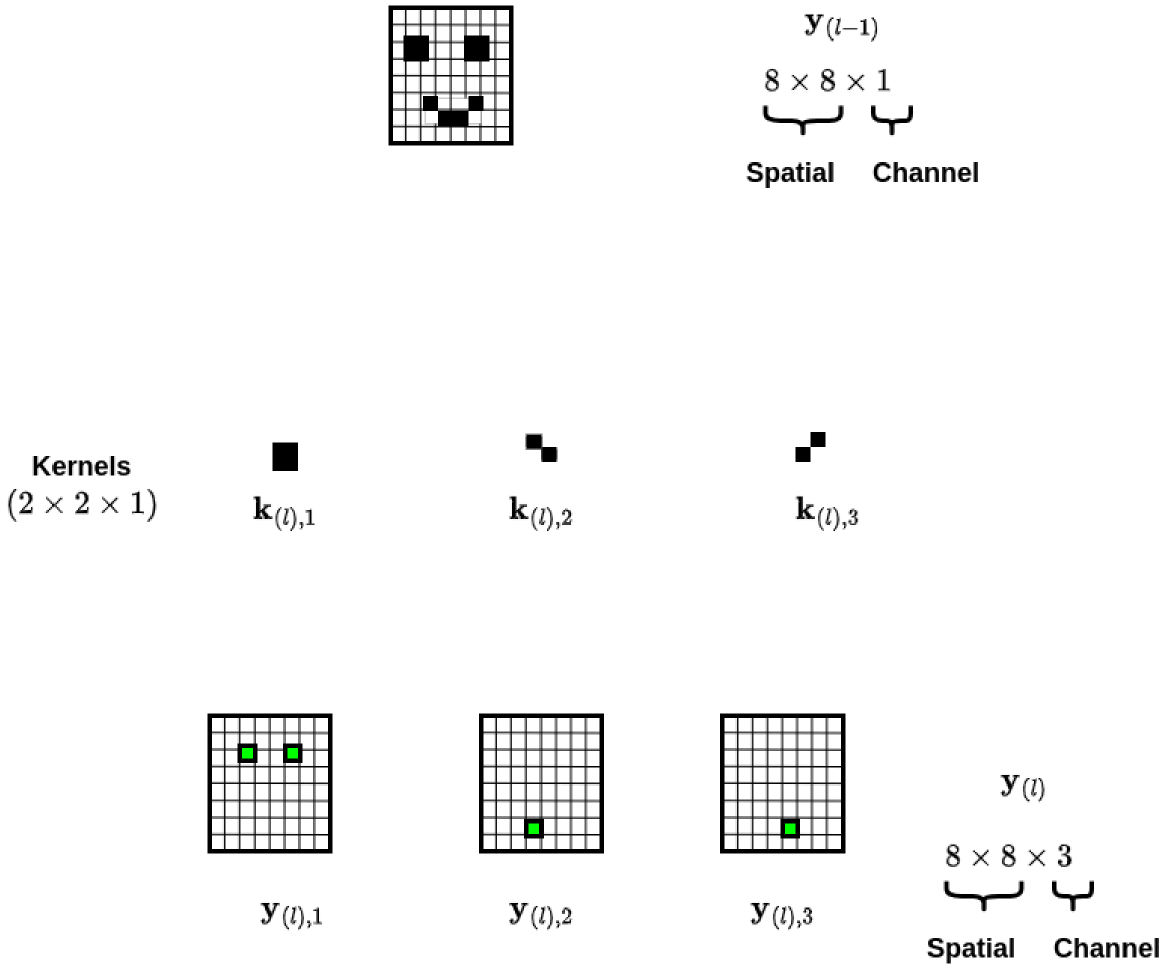
And, just as before

- we can use multiple patterns
- to get multiple output features

Each output feature $\mathbf{y}_{(l),j}$

- is a feature map for the j^{th} pattern

Convolution: 1 input feature to 3 output features



Hopefully you can see why small patterns are useful

- they result in feature maps that locate the presences of the small pattern
- at any location in the non-feature dimensions of the example

Convolution: detailed explanation

Define operation Conv with two arguments

- an example $\mathbf{x}$ with
  - N non-feature dimensions of lengths $d_1, \ldots d_N$
  - a feature dimension of length n
- a pattern $\mathbf{k}$ with
  - N non-feature dimensions of lengths $d'_1, \ldots d'_N$
    - such that
      $$d'_i \leq d_i \, : \, 1 \leq i \leq N$$
  - a feature dimension of length n

The $\mathbf{k}$ is able to be *smaller* than $\mathbf{x}$ in each non-feature dimensions.

Conv produces an output $\mathbf{y}$

- with N non-feature dimensions of lengths $d_1, \ldots d_N$
    - same as the N non-feature dimensions of $\mathbf{x}$
- a feature dimension of length 1

We define $\mathbf{y} = \text{Conv}(\mathbf{x}, \mathbf{k})$ by the output it produces

- at each index in the set of indexes of the elements in $\mathbf{x}$
  $$\text{idx} \in [1 : d_1] \times [1 : d_2] \times \ldots [1 : d_N]$$

Let

- $\text{SubRegion}(\mathbf{x}, \text{idx}, \mathbf{k})$
    - denote a sub-region of $\mathbf{x}$
    - centered at index idx
    - with non-feature dimensions identical to those of $\mathbf{k}$
    - and all n features

Then

$$\mathbf{y}_{\text{idx}} = \text{SubRegion}(\mathbf{x}, \text{idx}, \mathbf{k}) \cdot \mathbf{k}$$

So

- the output feature map $\mathbf{y}$
- has $d_1 * d_2 * \ldots d_N$ elements
- letting idx represent the index of just one element
- $\mathbf{y}_{\text{idx}}$ is the result of matching
    - pattern $\mathbf{k}$
    - with a the sub-region (of size matching $\mathbf{k}$) of the example
    - centered at idx

When can generalize this to an *multiple* patterns.

The above definition defines *one* feature map corresponding to the match against a single pattern.

In the presence of multiple patterns K

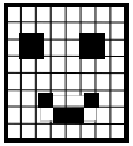
- output $\mathbf{y}$ has feature dimension of length K
- $\mathbf{y}_{\text{idx},j}$ is the feature map corresponding to the j^{th} pattern

The Convolutional Neural Network (CNN) layer type

We define a Neural Network Layer type

- called the *Convolutional Neural Network (CNN)* to implement the Conv operator against multiple patterns.

Convolution Layer: 1 input feature to 3 output features



$$\mathbf{y}_{(l-1)}$$

$$\underbrace{8 \times 8}_{\textbf{Spatial}} \times \underbrace{1}_{\textbf{Channel}}$$

In the diagram below we show a *Convolutional Layer*

- involving $\mathbf{3}$ "small" patterns

that is performed by a CNN Layer type that is layer l of a Sequential NN

- the input is $\mathbf{y}_{(l-1)}$ (the output of layer $(l-1)$ in a multi-layer NN)
- there are 3 patterns with non-spatial dimensions (2×2)
    - $\mathbf{k}_{(l),1}$ is the pattern for an "eye"
    - $\mathbf{k}_{(l),2}$ and $\mathbf{k}_{(l),3}$ are patterns for the left/right corner of the smile
- the output feature map $\mathbf{y}_{(l)}$ (the layer output)
    - has non-feature dimensions equal in number and length to those of $\mathbf{y}_{(l-1)}$
    - shows the locations within input $\mathbf{y}_{(l-1)}$ where the pattern is matched

Convolutional Neural Network (CNN) layer summary

Consider input layer $(l-1)$ with

- N spatial dimensions
- $n_{(l-1)}$ feature maps/channels
  $||\mathbf{y}_{(l-1)}|| = (d_{(l-1),1} \times d_{(l-1),2} \times \ldots d_{(l-1),N} \times n_{(l-1)})$

Convolutional Layer l will apply a Convolution that

- preserves the spatial dimensions
- but *may* change the number of features

$$||\mathbf{y}_{(l)}|| = (d_{(l-1),1} \times d_{(l-1),2} \times \ldots d_{(l-1),N} \times n_{(l)})$$

using $n_{(l)}$ kernels

- each of dimension
  $(f_{(l).1} \times f_{(l),2} \times \ldots f_{(l).N} \times n_{(l-1)})$

Typically:

$$f_{(l),i} = f_{(l)} \text{ (a constant) for } 1 \leq i \leq N$$

We summarize the key points

Reminders

Inputs

- The number N of non-feature dimensions
    - of the kernel and example are the same
- The *length* of each non-feature dimension of the kernel
    - is less than or equal to the length of the corresponding dimension of the example
- The feature dimension's length of the kernel and example are the same

Outputs

- The number of output features equals the number of kernels
- The length of the output's non-feature dimensions is the same as the length of the corresponding dimension of the example
    - this is true only with "same" padding
    - without full padding: $\lfloor \frac{f}{2} \rfloor$ elements may be lost from each end of a dimension
        - where f is length of each kernel non-feature dimension

Convolution (with full padding)

- changes the length of the feature dimension

Key point

A Convolutional Layer

- preserves *non-feature* dimensions (assuming "same" padding)
- **changes** the number of features from $n_{(l-1)}$ to $n_{(l)}$

How is the Feature dimension different from non-feature dimensions ?

The feature dimension has some key differences from the non-feature dimensions

- the indices of the feature dimension are *unordered*
    - permuting the features
        - from `Price, Volume, Open, Close`
        - to `Open, Close, Price, Volume`
    - does not change the meaning of the example

In contrast the *non-feature* dimensions are at least *partially ordered*

- permuting the order of a non-feature dimensions *changes the meaning* of an example

Consider

- the indices of the temporal dimension are totally ordered
    - reversing the indices makes time flow backwards rather than forwards
- the indices of the spatial dimension are (at least, partially) *ordered*
    - given an image of a face
        - the eyes are located above the mouth
        - in a horizontal orientation

- words in a sentence are ordered

$$\begin{aligned}\mathbf{x} &= [\text{Machine, Learning, is, easy, not, hard }] \\ \mathbf{x}[\text{perm}] &= [\text{Machine, Learning, is, hard, not, easy}]\end{aligned}$$

    - very different meanings

Convolution respects the relative order of the elements of the example.

Layers that operate purely on the feature dimensions do not respect the order of features.

For example

- A `Dense` layer will compute the *same dot-product* of features and weights
- as long they both obey the same order

$$\mathbf{x} \cdot \mathbf{w} = \mathbf{x}[\text{perm}] \cdot \mathbf{w}[\text{perm}]$$

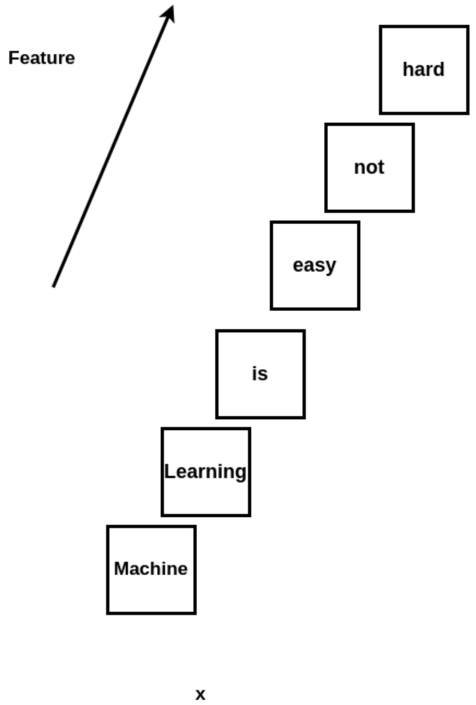
even though $\mathbf{x}$ and $\mathbf{x}[\text{perm}]$ have much different meanings.

That is: order is not respected by the feature dimension.

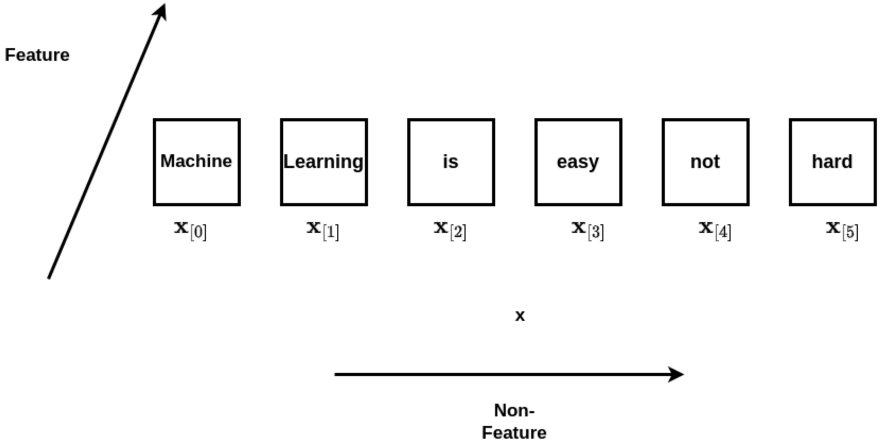
It is generally problematic to try to use the feature dimension as a replacement for a non-feature dimensions.

- when ordering is important

Feature only vector



One feature plus one temporal dimension



Where do the patterns come from ? Training a CNN

Hopefully you understand how patterns (kernels) are "feature recognizers".

But you may be wondering: how do we determine the weights in each kernel ?

Answer: a Convolutional Layer is "just another" layer in a multi-layer network

- The kernels are just weights (like the weights in Fully Connected layers)
- We solve for all the weights $\mathbf{W}$ in the multi-layer network in the same way

The answer is: exactly as we did in Classical Machine Learning

- Define a loss function that is parameterized by $\mathbf{W}$:
$$\mathcal{L} = L(\hat{\mathbf{y}}, \mathbf{y}; \mathbf{W})$$

- The kernel weights are just part of $\mathbf{W}$

- Our goal is to find $\mathbf{W}^*$ the "best" set of weights
$$\mathbf{W}^* = \operatorname*{argmin}_W L(\hat{\mathbf{y}}, \mathbf{y}; \mathbf{W})$$
- Using Gradient Descent !

In other words: their is nothing special about finding the "best" kernels.

Conv1d: the under-appreciated convolution

CNN's seem to be most associated with *image* input

- Two non-feature dimensions

It's worth pointing out how a one-dimensional (single non-feature dimension) convolution an be used.

Time Series

Let the single non-feature dimensions denote time.

- I will use subscript t to index into the elements

Suppose

- Example $n = 1$ features: `Volume`
- single kernel
    - $f = 3$: length of kernel

So the output $\mathbf{y}$ will

- be a timeseries of length equal to the example
- have a single feature
    - one kernel

Then by definition of convolution (writing out the dot product for each index)

$$\mathbf{y}_{(t),1} = \sum_{o=-1}^{o=+1} \mathbf{x}_{(t+o,)1} * \mathbf{k}_{o+1,1}$$

The convolution is just a *moving average* !

- with *learned* weights: the kernel values

Note the subscript "1" to refer to the single feature.

A word of warning

- $\mathbf{y}_{(t),1}$ references a value that occurs after time t
    - $\mathbf{y}_{(t+1),1}$

Depending on your task

- this may be dis-allowed
- equivalent to "peeking into the future"

Causal convolution is a restriction on convolution to prevent looking ahead into the future.

NLP: n-grams

We have not yet covered Natural Language Processing (NLP)

- but we can give some intuition
- on how one dimensional Convolution may be used

Since words (really: tokens) are *categorical* values

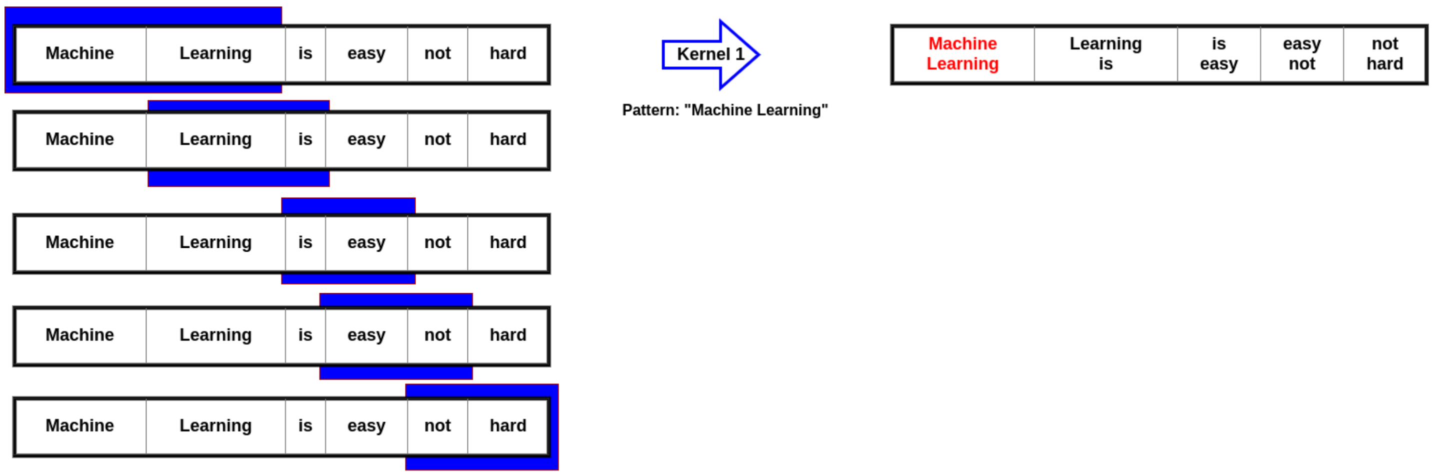
- we need to "numericalize" them
- turn into vectors
    - OHE vectors, of length equal to number of tokens in vocabulary
    - dense vectors: Embeddings

We use n to denote the number of features of the vector that encode tokens.

Below we illustrate

- example that is a sequence of words
    - elements are indexed by time/position
- n is length of the vector encoding of a token
- Single Kernel
    - $f = 2$
- No padding
    - so output sequence loses one element at start of sequence

NLP: Conv1d
single kernel



The one-dimensional convolution

- coverts two consecutive tokens
- into a single vector

This is called creating a *bi-gram*

- if we combine f tokens, we call it an *f-gram* f-grams are interesting and useful because

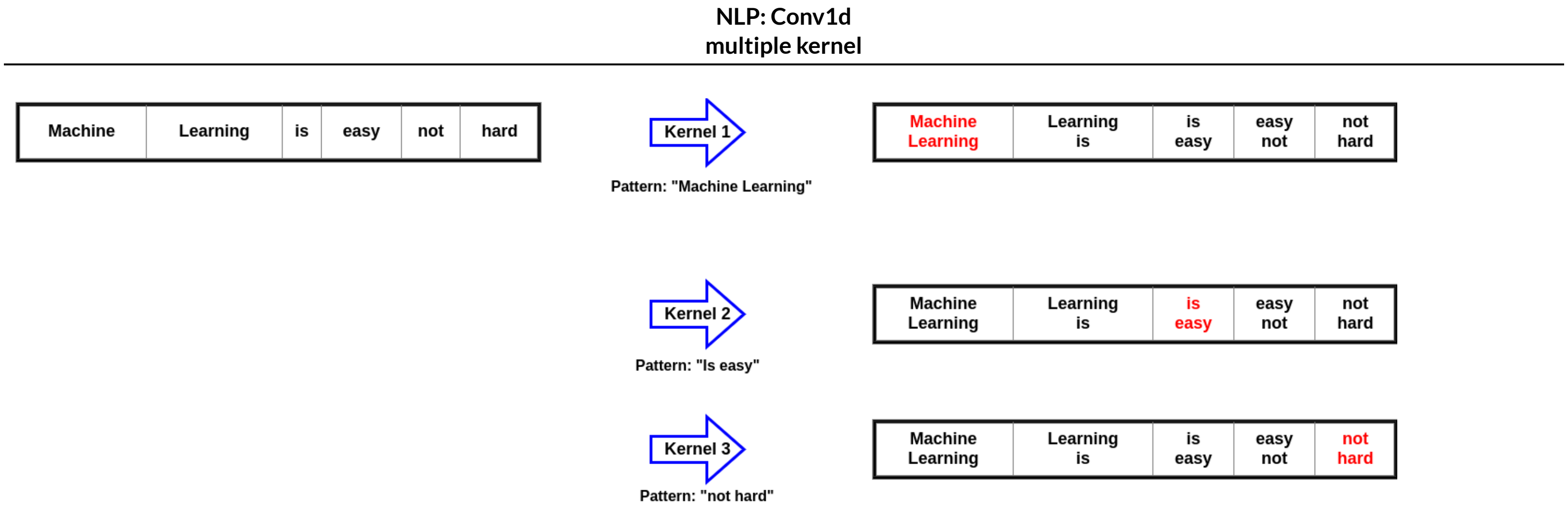
- sometimes f consecutive tokens
- denote a new *semantically meaningful* concept
- that is very different from the individual tokens

For example

- "Machine Learning"
- "New York City"

Below we expand this to multiple output features

- each pattern is "looking for" (has maximum scalar dot product)
- the concept highlighted in red



In [4]:

```
print("Done")
```

```
Done
```