Recurrent nets with keras

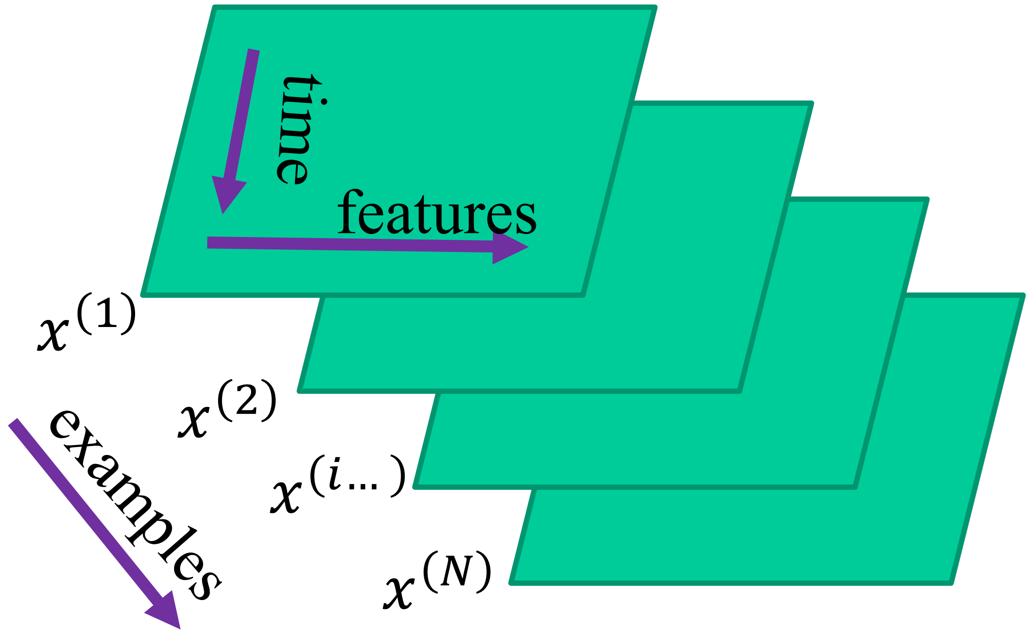
Professor Marie Roch

SAN DIEGO STATE UNIVERSITY

Sequences in keras

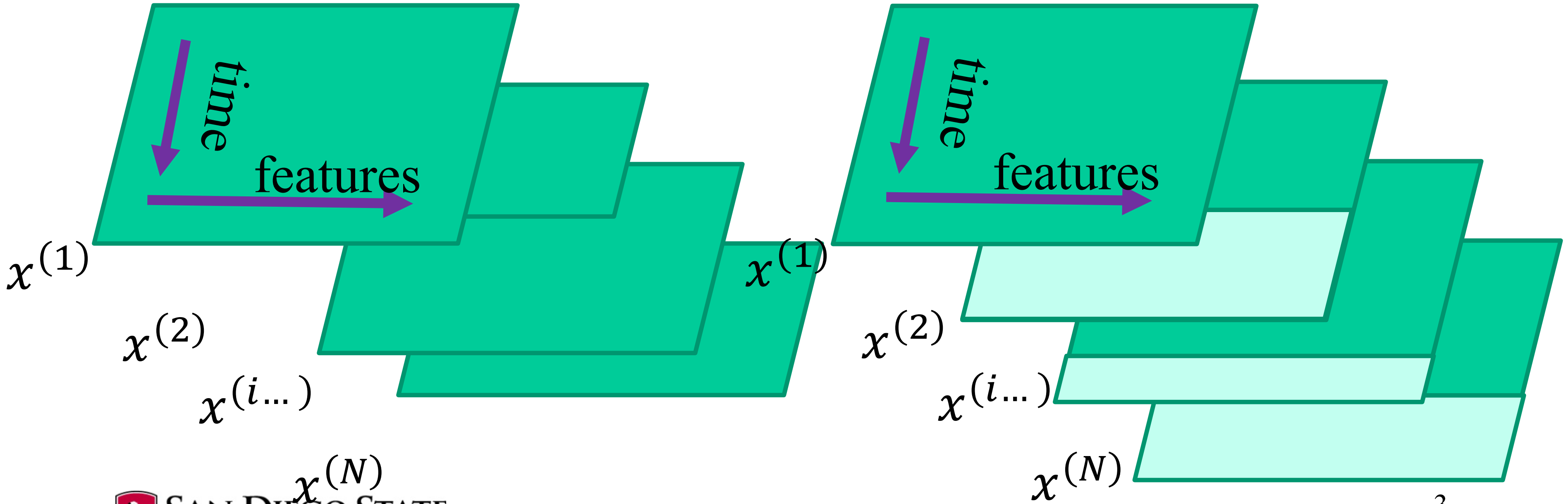
- Input sequences are tensors
- Each tensor has the following shape:

$$example \times time \times dim$$



Varying length sequences

- Sequences vary in length, tensors do not
- Pad tensor a standard length (e.g., 0 fill)



Varying length sequences

- Padding can be expensive
  - A few long sequences make everything of that length
- Solution: use a mini-batch generator, e.g.
  - Keras Sequence
  - Tensorflow dataset

Input layer

- The input layer should have dimension (None, dim) –arbitrary # of fixed dimension vectors

- Masking layer can be used to tell RNN to ignore time steps with a specified value (“mask_value”:*constant_value*)

Feed forward layers on time-series

- If feed-forwards layers are desired prior to the recurrent layer, use a layer wrapper, e.g.:

  TimeDistributed(Dense(40))

- This does the following:
  - pass time slices one by one through Dense layer
  - reconstruct a tensor to be used by the next layer

Recurrent layers

- Recurrent layers (e.g. LSTM, GRU) are added like any other layer and can be followed by dropout layers.

Recurrent layer options

- recurrent_regularizer: Allows specification of regularizer for the recurrent weights
- return_sequences
  - True – A sequence of outputs is generated
  - False – Only the last output is returned. Appropriate when a decision is to be made based on the state at the end of the sequence

LSTM options

- return_state – If True, returns the unit state in addition to the output
- go_backwards – If True, dependent on future inputs
- unroll – If True, network is unrolled. Faster, but inappropriate for long sequences

LSTM options

- stateful – If True, subsequent batches have a continuation of the current state for each example
  e.g. N examples that are very long
  - Last input of example 3 batch 1 continued by example 3 of batch 2 (appropriate for long time-series)

Many to 1 classification

- To classify a sequence, use return_sequences=False on last recurrent layer.
- Then add feed-forward layers as appropriate

Many to many classification

- When a recurrent layer returns a sequence, we may want subsequent feed forward layers (e.g. softmax or something more complicated)
- Wrap subsequent feed-forward layers in a TimeDistributed layer

Paradigm for recurrent network construction from data structures

- Similar to what we used before
  `(Layer, [positional args], {dict of keyword args})`
- Tuples in data structure have an optional 4th element to permit wrappers:

```
(Dense, [output_classes_N],
       {'activation':'softmax',
        'kernel_regularizer':regularizers.l2(l2)},
        # The Dense layer is not recurrent, we need to wrap it in
        # a layer that that lets the network handle the fact that
        # our tensors have an additional dimension of time.
        (TimeDistributed, [], {}))
```

- Cannot be used for complex networks
  (e.g. U-net architectures)

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Masking, Dense, …

def build_model(specification, name="model"):
    """build_model - specification list
    Create a model given a specification list
    Each element of the list represents a layer and is formed by a tuple.

    (layer_constructor,
     positional_parameter_list,
     keyword_parameter_dictionary)

    Example, create M dimensional input to a 3 layer network with
    20 unit ReLU hidden layers and N unit softmax output layer

    [(Dense, [20], {'activation':'relu', 'input_dim': M}),
     (Dense, [20], {'activation':'relu', 'input_dim':20}),
     (Dense, [N], {'activation':'softmax', 'input_dim':20})
    ]
```

```
Wrappers are supported by creating a 4th item in the tuple/list
that consists of a tuple with 3 items:
    (WrapperType, [positional args], {dictionary of arguments})

The WrapperType is wrapped around the specified layer which is assumed
to be the first argument of the constructor.  Additional positional
argument are taken from the second item of the tuple and will *follow*
the wrapped layer argument.  Dictionary arguments
are applied as keywords.

For example:
(Dense, [20], {'activation':'relu'}, (TimeDistributed, [], {}))

would be equivalent to calling TimeDistributed(Dense(20, activation='relu'))
If TimeDistributed had positional or named arguments, they would be placed
inside the [] and {} respectively.  Remember that the wrapped layer (Dense)
in this case is *always* the first argument to the wrapper constructor.
"""
```

```
K.name_scope(name)
model = Sequential()

for item in specification:
    layertype = item[0]
    # Construct layer and add to model
    layer = layertype(*item[1], **item[2])

    if len(item) > 3:
        wrapspec = item[3] # User specified wrapper
        # Get type, positional args and named args
        wraptype, wrapposn, wrapnamed = wrapspec
        wlayer  = wraptype(layer, *wrapposn, **wrapnamed)
        model.add(wlayer)
    else:
        # No wrapper, just add it.
        model.add(layer)

return model
```

Cross validation

- Scikit: sklearn.model_selection.Kfold
- Constructor: Kfold(# splits, shuffle)
- Example:

```
crossval = Kfold(10, shuffle=True)
for (train, test) in crossval.split(devex, devlab):
        train_model(devex)
```