Limitations of recurrent neural networks

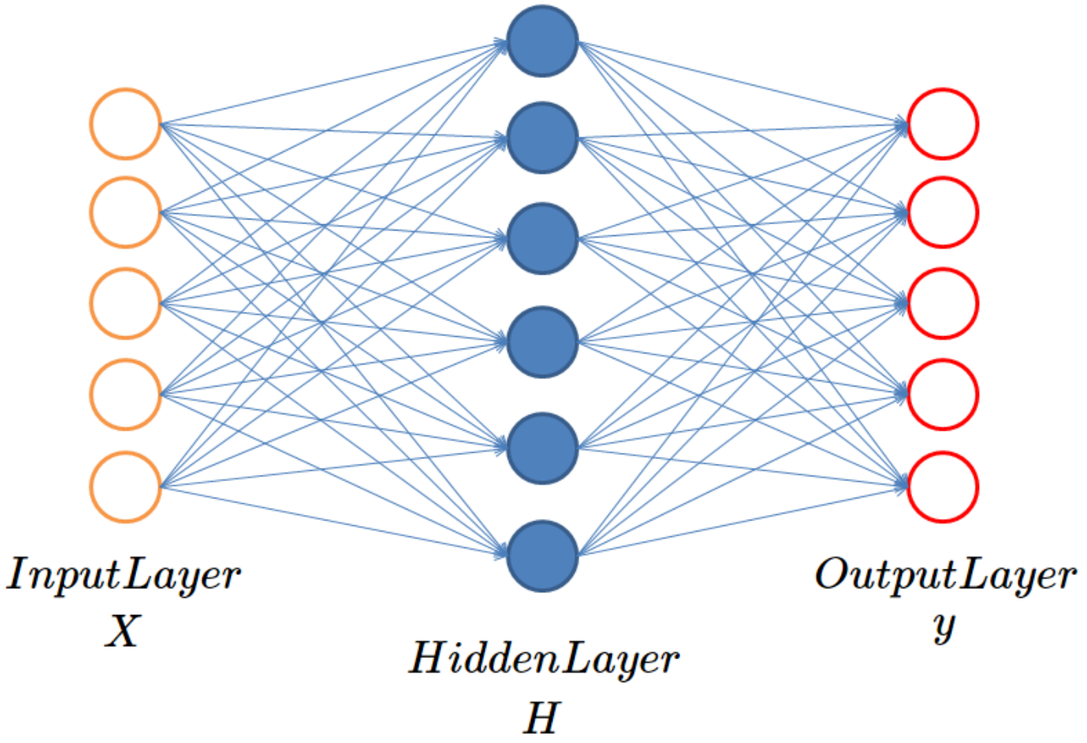
NATURAL LANGUAGE GENERATION IN PYTHON

Biswanath Halder
Data Scientist

datacamp

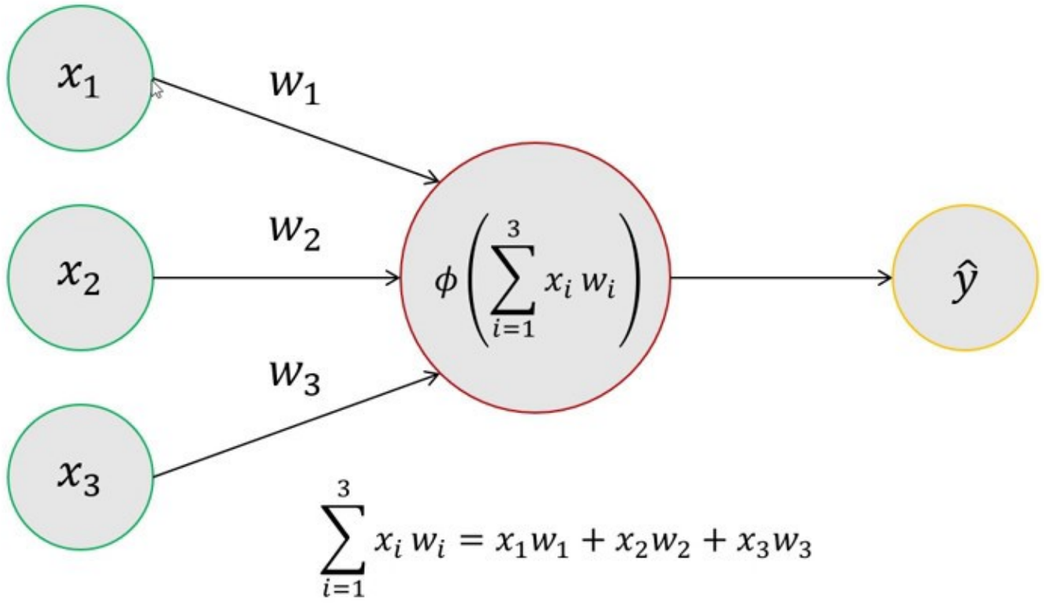
Simple neural networks

- Nodes arranged in layers.
- Nodes in different layers connected by weights.
- Input to first layer : data.
- Input to other layers : output from previous layer.



Computations in neural network

- Linear transformation.
- Followed by non-linear transformation.
- Linear followed by non-linear transformation makes network powerful.



Gradient and training

- Error: squared difference of actual output and predicted output.

$$E = \sum e_i = \sum (y_i - \hat{y}_i)^2$$

- Gradient: Rate of change of error with respect to weights.

$$g_i = \Delta E / \Delta w_i = \partial E / \partial w_i$$

- Training: Adjusting weights to reduce error.

$$w_i = w_i - \eta * \partial E / \partial w_i$$

- Learning rate (η): factor by which to adjust the weights.

Chain rule

- Chain rule

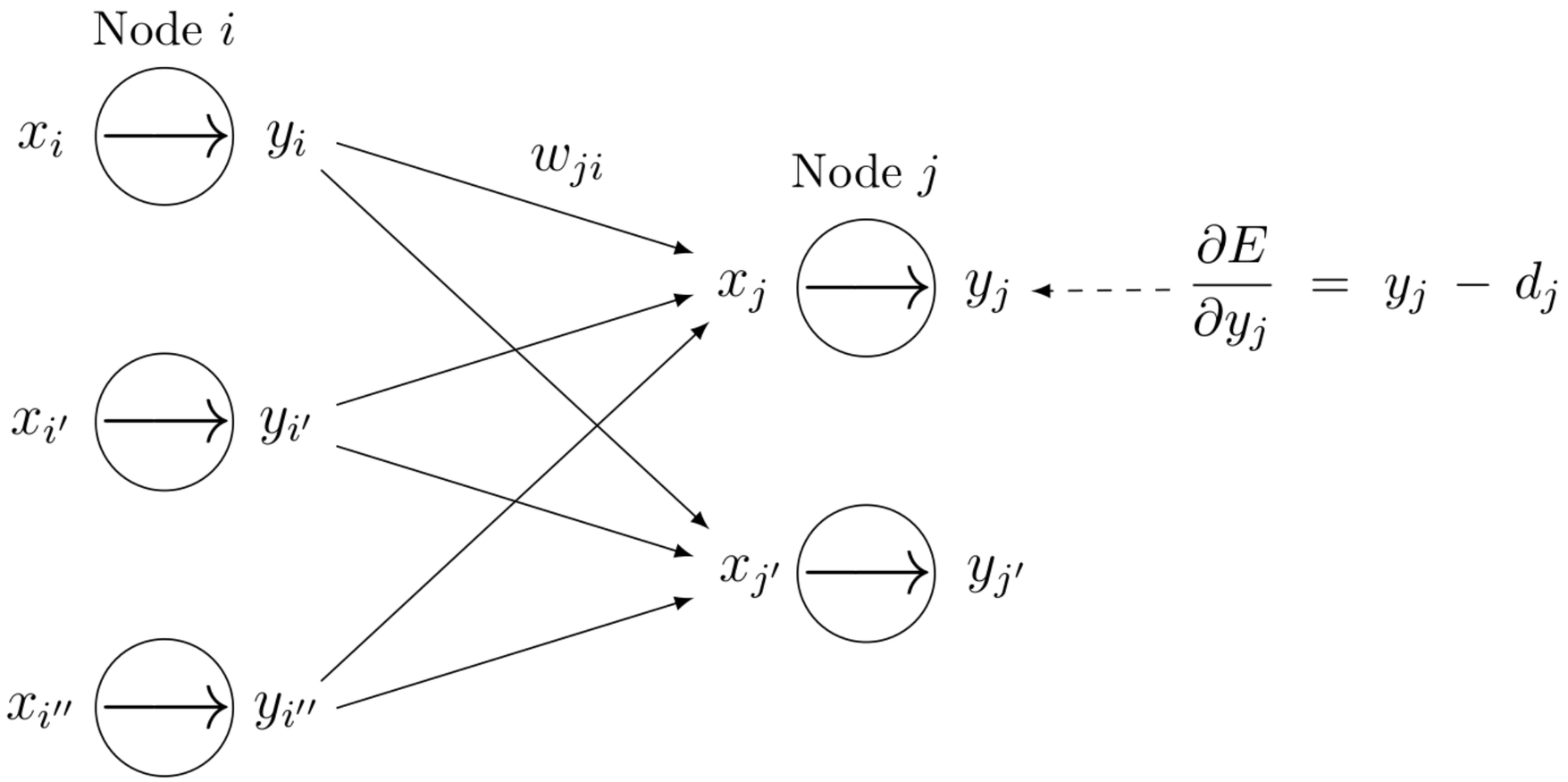
$$z = g(y), y = f(x)$$

$$\partial z/\partial x = (\partial z/\partial y) * (\partial y/\partial x)$$

.

- Gradients in output layer found by differentiation.
- For other layers, chain rule applied.

Back-propagation



Vanishing and exploding gradients

- Gradient: product of many gradient values from subsequent time-steps.
- Gradients calculated at output layer.
- Gradients propagated back from next time-step to previous.
- Vanishing gradients: gradients become smaller in earlier time-steps.
- Exploding gradients: gradients become bigger as you back-propagate.

Remedies

- Vanishing gradients: fixed number of time-steps for back-propagation.
- Exploding gradients: gradient clipping
- Results in suboptimal training and reduced performance.

Let's practice!

NATURAL LANGUAGE GENERATION IN PYTHON

Introduction to long short term memory

NATURAL LANGUAGE GENERATION IN PYTHON

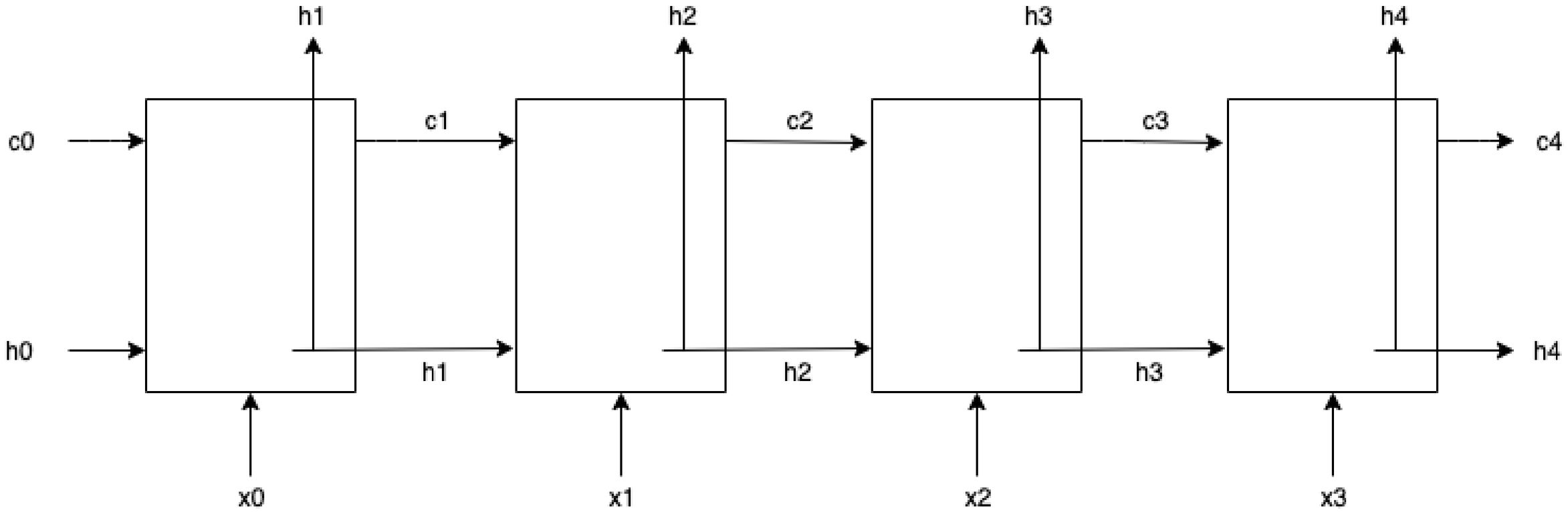
Biswanath Halder
Data Scientist

datacamp

Long-term dependencies

- Short-term dependency: "The birds are flying in the sky".
- Long-term dependency: "I was born in Germany.....I can speak German".
- RNN good for short-term dependency.
- RNN can't handle long-term dependencies well.

Long-short term memory



Write like Shakespeare

- Input: Dataset of selected works of Shakespeare in text format.
- Goal: Generate text imitating Shakespeare's unique writing style.
- Get the vocabulary.

```
vocabulary = sorted(set(text))
```

- Character to integer and the reverse mapping.

```
char_to_idx = dict((char, idx) for idx, char in enumerate(vocabulary))
idx_to_char = dict((idx, char) for idx, char in enumerate(vocabulary))
```

Input and target data

- Sentence: "I may contrive our father; and, in their defeated queen"
- Input : "I may contrive our father; and, in their defeated quee"
- Output: "n"

Input and target data from raw text

```
input_data = []
target_data = []
maxlen = 40
for i in range(0, len(text) - maxlen):
    # Find the sequence of length maxlen starting at i
    input_data.append(text[i : i + maxlen])

    # Find the next char after this sequence
    target_data.append(text[i + maxlen])
```

Create input and target vectors

- Vector to encode input data

```
# Create a 3-D zero vector to contain the encoded input sequences
x = np.zeros((len(input_data), maxlen, len(vocabulary)), dtype='float32')
```

- Vector to encode output data

```
# Create a 2-D zero vector to contain the encoded target characters
y = np.zeros((len(target_data), len(vocabulary)), dtype='float32')
```

Initialize input and target vector

- Fill the input and target vectors with data.

```
# Iterate over the sequences
for s_idx, sequence in enumerate(input_data):
    # Iterate over all characters in the sequence
    for idx, char in enumerate(sequence):
        # Fill up vector x
        x[s_idx, idx, char_to_idx[char]] = 1
    # Fill up vector y
    y[s_idx, char_to_idx[target_data[i]]] = 1
```

Create LSTM network in Keras

- Create sequential model.

```
model = Sequential()
```

- Add LSTM layer.

```
model.add(LSTM(128, input_shape=(maxlen, len(vocabulary))))
```

- Add Dense output layer.

```
model.add(Dense(len(vocabulary), activation='softmax'))
```

Compile the model

- Compile the model.

```
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

- Check model summary.

```
model.summary()
```

Let's practice!

NATURAL LANGUAGE GENERATION IN PYTHON

Inference using long short term memory

NATURAL LANGUAGE GENERATION IN PYTHON

Biswanath Halder
Data Scientist

datacamp

Training and validation

- Training: error reduction on the training set.
- Ensures good performance on training data.
- Doesn't ensure good performance on unseen data.
- Separate sample to test performance on new data.
- Test or validation set.

The LSTM model

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 128)               84480
_________________________________________________________________
dense_1 (Dense)              (None, 36)                4644
=================================================================
Total params: 89,124
Trainable params: 89,124
Non-trainable params: 0
_________________________________________________________________
```

Training LSTM model

- Fit LSTM model.

```
model.fit(x, y, batch_size=64, epochs=1, validation_split=0.2)
```

- Batch size: number of samples after which weights gets adjusted.
- Epoch: number of times to iterate over the full dataset.
- Validation split: percentage of samples kept aside for test set.

Seed sequence for prediction

- Seed sentence.

```
sentence = "that, poor contempt, or claim'd thou sle"
```

- Encoded sentence.

```
X_test = np.zeros((1, maxlen, len(vocabulary)))
for t, char in enumerate(sentence):
    X_test[0, t, char_to_idx[char]] = 1.
```

Predict the next character

- Probability distribution for the next character.

```
preds = model.predict(X_test, verbose=0)
prob_next_char = preds[0]
```

- Index with highest probability.

```
next_index = np.argmax(prob_next_char)
```

- Mapping the index to the actual character.

```
next_char = idx_to_char[next_index]
```

```
def generate_text(sentence, n):
    generated += sentence
    for i in range(n):
        # Create input vector from the input sentence
        x_pred = np.zeros((1, maxlen, len(vocabulary)))
        for t, char in enumerate(sentence):
            x_pred[0, t, char_to_idx[char]] = 1.

        # Get probability distribution for the next character
        preds = model.predict(x_pred, verbose=0)[0]

        # Get the character with maximum probability
        next_index = np.argmax(preds)
        next_char = idx_to_char[next_index]

        # Append the new character to the next input and generated text
        sentence = sentence[1:] + next_char
        generated += next_char

    # Print the generated text
    print(generated)
```

Generate text imitating Shakespeare

```
that, poor contempt, or claim'd thou sleds,
and some in the shart no or me the goonl
for i am the starn his more in fone, wether read,
and thou art and summon the self a love,
that thou mure thou muer of shakes hide.
the earth shall of your love shall nother,
which thou thy sweet ont not be nother love.
for the shart my wids hatust thou mure,
when it thy sweet breathing of seart, doth stather decest,
and thou art and summonied it how thy sell,
the earth sweet beauty, by nothers dece.
the earth shall of me thou mu
```

Let's practice!

NATURAL LANGUAGE GENERATION IN PYTHON