Rekurencyjne sieci neuronowe

dr inż. Sebastian Ernst

Przedmiot: Uczenie Maszynowe

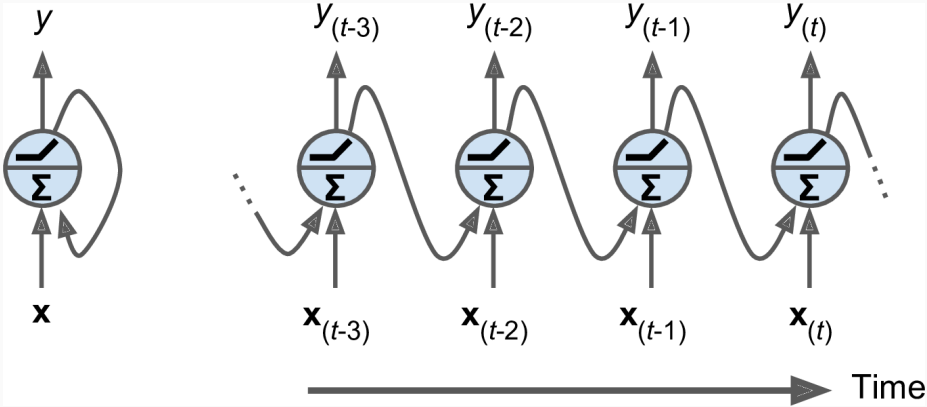
Rekurencyjne sieci neuronowe

Przetwarzanie szeregów czasowych

- najczęściej w celu przewidywania przyszłości
- zastosowania
  - finanse (giełda)
  - pojazdy autonomiczne
  - sterowanie
  - wykrywanie usterek

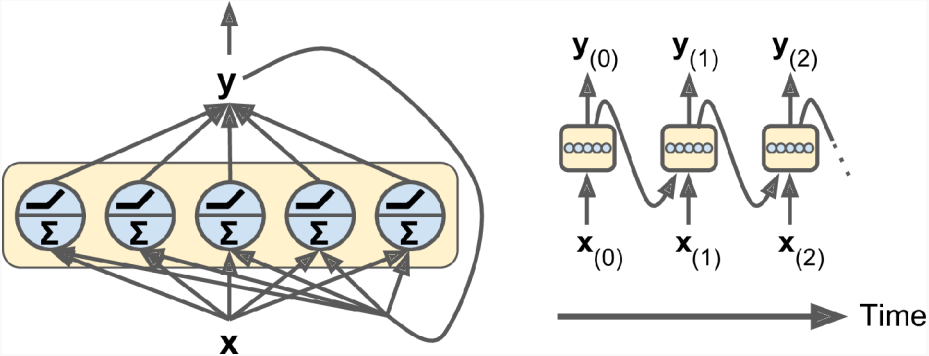
Rekurencyjne neurony

- otrzymują sygnały ze swoich poprzednich wyjść
- krok czasowy t = *ramka*
- rozciągnięcie w czasie – *unrolling*



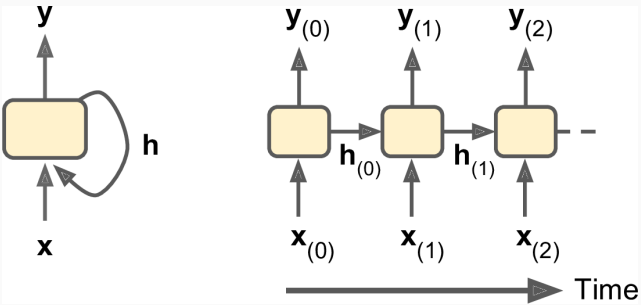
Warstwa rekurencyjnych neuronów

- otrzymuje wektor wejściowy *oraz* wektor wyjściowy z poprzedniego kroku
- każdy neuron ma dwa zestawy wag

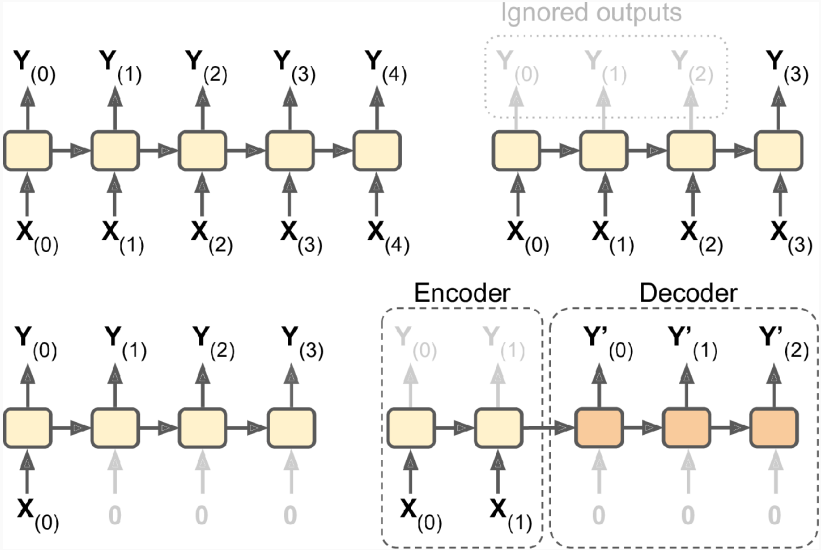


Komórki pamięci

- wyjście w chwili t zależy od wejść w poprzednich chwilach
- neuron ma więc rodzaj *pamięci*
- typowo pamięć jest krótka (ok. 10 kroków)
- wyjście z komórki może nie być tożsame z jej stanem wewnętrznym

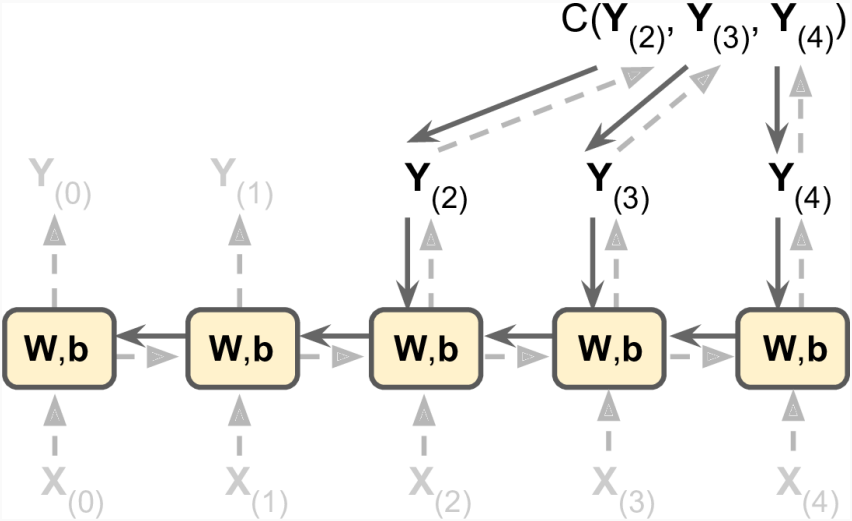


Sekwencje wejściowe i wyjściowe



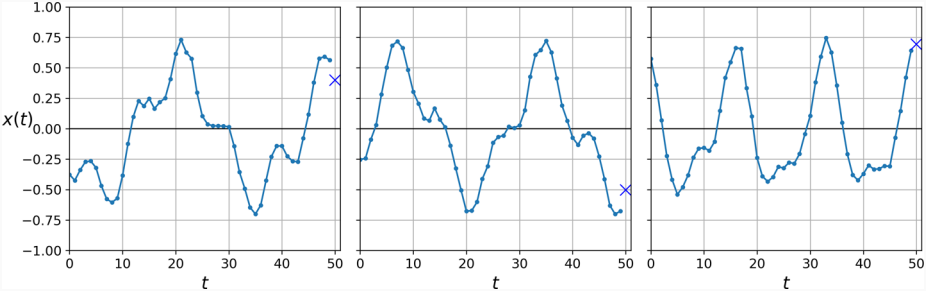
Uczenie RNN

- *unrolling* i zwykła propagacja wsteczna = BPTT (*backpropagation through time*)



Predykcja szeregów czasowych

- przewidywanie – *forecasting*
- wypełnianie luk w przeszłości – *imputation*



Generujemy szeregi czasowe

```
def generate_time_series(batch_size, n_steps):
    freq1, freq2, offsets1, offsets2 = np.random.rand(4, batch_size, 1)
    time = np.linspace(0, 1, n_steps)
    series = 0.5 * np.sin((time - offsets1) * (freq1 * 10 + 10))  #   wave 1
    series += 0.2 * np.sin((time - offsets2) * (freq2 * 20 + 20)) # + wave 2
    series += 0.1 * (np.random.rand(batch_size, n_steps) - 0.5)   # + noise
    return series[..., np.newaxis].astype(np.float32)
```

Wskaźniki bazowe

Predykcja naiwna:

```
y_pred = X_valid[:, -1]
```

Predykcja siecią gęstą – model liniowy:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[50, 1]),
    keras.layers.Dense(1)
])
model.compile(loss="mse", optimizer="adam")
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
model.evaluate(X_valid, y_valid)
```

Najprostsza RNN

Jeden neuron, jedna warstwa:

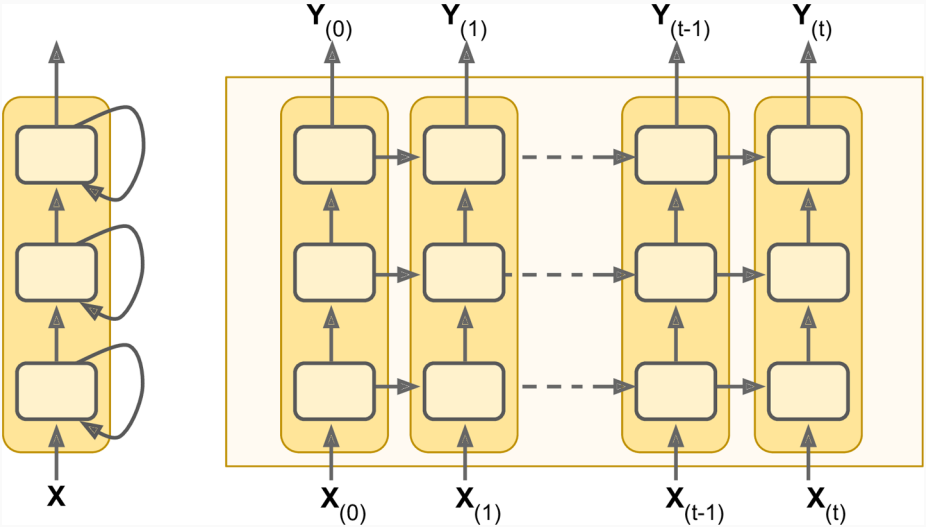
```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(1, input_shape=[None, 1])
])

optimizer = keras.optimizers.Adam(learning_rate=0.005)
model.compile(loss="mse", optimizer=optimizer)
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
```

Ta sieć ma łącznie 3 parametry, a „liniowa” sieć gęsta – 51.

Głębokie RNN

Głęboka RNN



Głęboka RNN, przykład

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True,
                           input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.SimpleRNN(1)
])

model.compile(loss="mse", optimizer="adam")
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
```

- ukryty stan w warstwie wyjściowej to zaledwie jedna liczba
- warstwy RNN korzystają z *tanh*, a więc wyjście musi być w zakresie $[-1, 1]$.

Głęboka RNN, drugie podejście

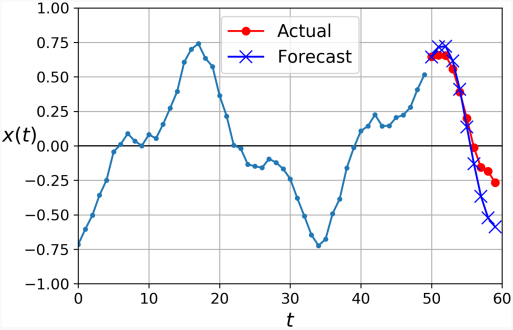
```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True,
                           input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(1)
])

model.compile(loss="mse", optimizer="adam")
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
```

- szybciej zbieżna
- można użyć dowolnej funkcji aktywacji

Prognozowanie kilku kroków naprzód

- pierwsze podejście: „doklejanie” predykowanych wartości na końcu szeregu
- drugie podejście: predykcja n wartości jednocześnie



Prognozowanie wielu kroków, przygotowanie danych

```
n_steps = 50
series = generate_time_series(10000, n_steps + 10)
X_train, Y_train = series[:7000, :n_steps], series[:7000, -10:, 0]
X_valid, Y_valid = series[7000:9000, :n_steps], series[7000:9000, -10:, 0]
X_test, Y_test = series[9000:, :n_steps], series[9000:, -10:, 0]
```

Prognozowanie wielu kroków, model

Zmieniamy liczbę neuronów w warstwie wyjściowej:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True,
                           input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(10)
])

model.compile(loss="mse", optimizer="adam")
history = model.fit(X_train, Y_train, epochs=20,
                    validation_data=(X_valid, Y_valid))
```

Model *sequence-to-sequence*

- uczymy model aby zwracał 10 wartości w każdym kroku, nie tylko w ostatnim
- przygotowujemy odpowiednio sekwencje etykiet (*target*)
- warstwy RNN muszą mieć argument `return_sequences=True`
- warstwę wyjściową uruchamiamy dla każdego kroku czasowego przy pomocy warstwy `TimeDistributed`
- do ewaluacji przyda się funkcja licząca MSE w ostatnim kroku

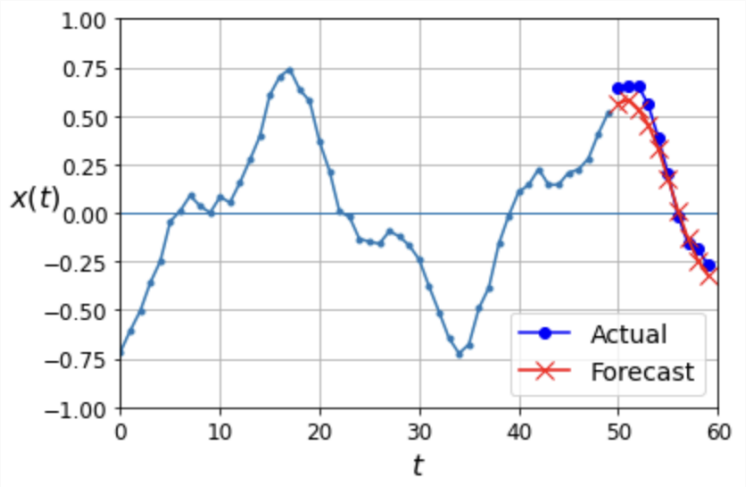
Model *sequence-to-sequence*, implementacja

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True,
                           input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])

def last_time_step_mse(Y_true, Y_pred):
    return keras.metrics.mean_squared_error(Y_true[:, -1],
                                            Y_pred[:, -1])

model.compile(loss="mse", optimizer=keras.optimizers.Adam(
    learning_rate=0.01), metrics=[last_time_step_mse])
history = model.fit(X_train, Y_train, epochs=20,
                    validation_data=(X_valid, Y_valid))
```

Model *sequence-to-sequence*, wyniki



Przetwarzanie długich sekwencji

Wyzwania

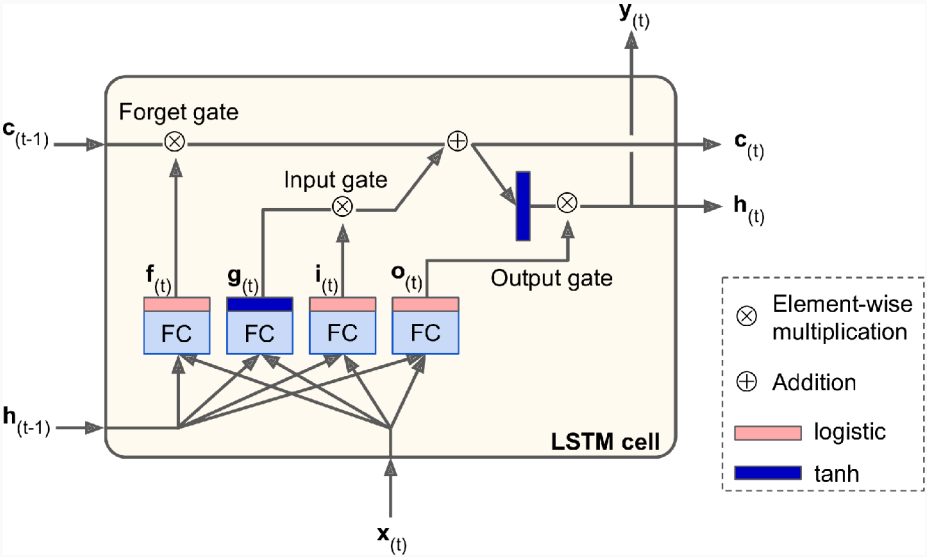
- wiele kroków, a więc sieć *unrolled* robi się bardzo głęboka
- może to powodować problemy niestabilności gradientu
- po pewnym czasie sieć „zapomina” początek sekwencji
- istnieją rodzaje komórek z pamięcią długoterminową

Komórki *Long Short-Term Memory* (LSTM)

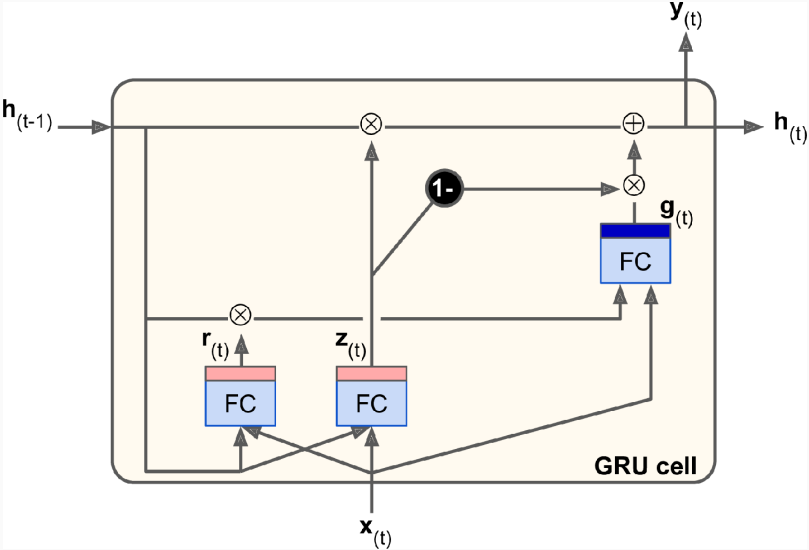
- 1997, Sepp Hochreiter, Juergen Schmidhuber
- użycie analogiczne jak przy prostych komórkach rekurencyjnych
- w Keras stosujemy warstwę `LSTM` (lub `RNN` z argumentem `LSTMCell` – ale bez optymalizacji GPU)

```
model = keras.models.Sequential([
    keras.layers.LSTM(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.LSTM(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

Budowa komórki LSTM



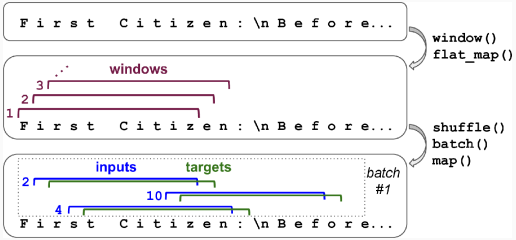
Komórki *Gated Recurrent Unit* (GRU)



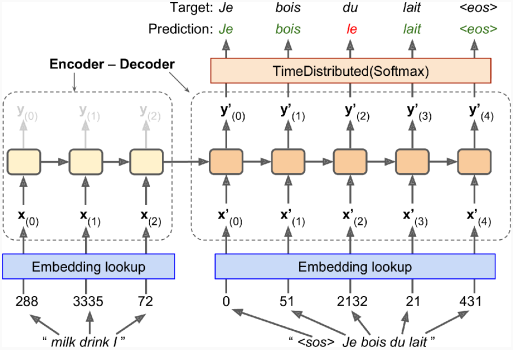
Przetwarzanie tekstu przy pomocy RNN

Przykład: generowanie tekstu

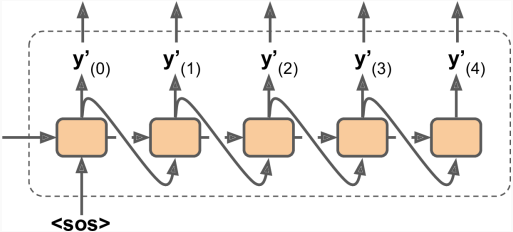
- Przetwarzamy tekst przy pomocy `tf.keras.layers.TextVectorization`
- Cechy i etykiety są fragmentami tekstu przesuniętymi o jeden znak:
  - *to be or not to b*
  - *o be or not to be*



Przykład: tłumaczenie przy pomocy enkodera-dekodera

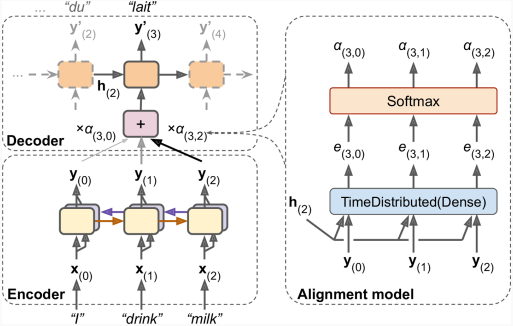


Na etapie predykcji (po treningu) nie mamy do dyspozycji tekstu docelowego, więc podajemy na wejście słowo wygenerowane poprzednio przez model:



Mechanizm *attention*

- Pozwala dekoderowi skupić się na odpowiednich słowach kodowanych przez enkoder.
- Z wyjść enkodera wyliczana jest średnia ważona.
- Wagi wyznaczane są przez małą sieć neuronową, zwaną *modelem wyrównującym* (ang. *alignment model*) lub *warstwą uwagi* (ang. *attention layer*).



Transformer: Attention is All You Need

- W 2017 zaproponowano architekturę bez CNN czy RNN, wyposażoną tylko w mechanizm *attention*.
- Model nie jest rekurencyjny, więc nie ma eksplozji głębokości po *unroll* oraz problemów z nią związanych.