VL Deep Learning for Natural Language Processing

13. Long Short-Term Memory

Prof. Dr. Ralf Krestel

AG Information Profiling and Retrieval

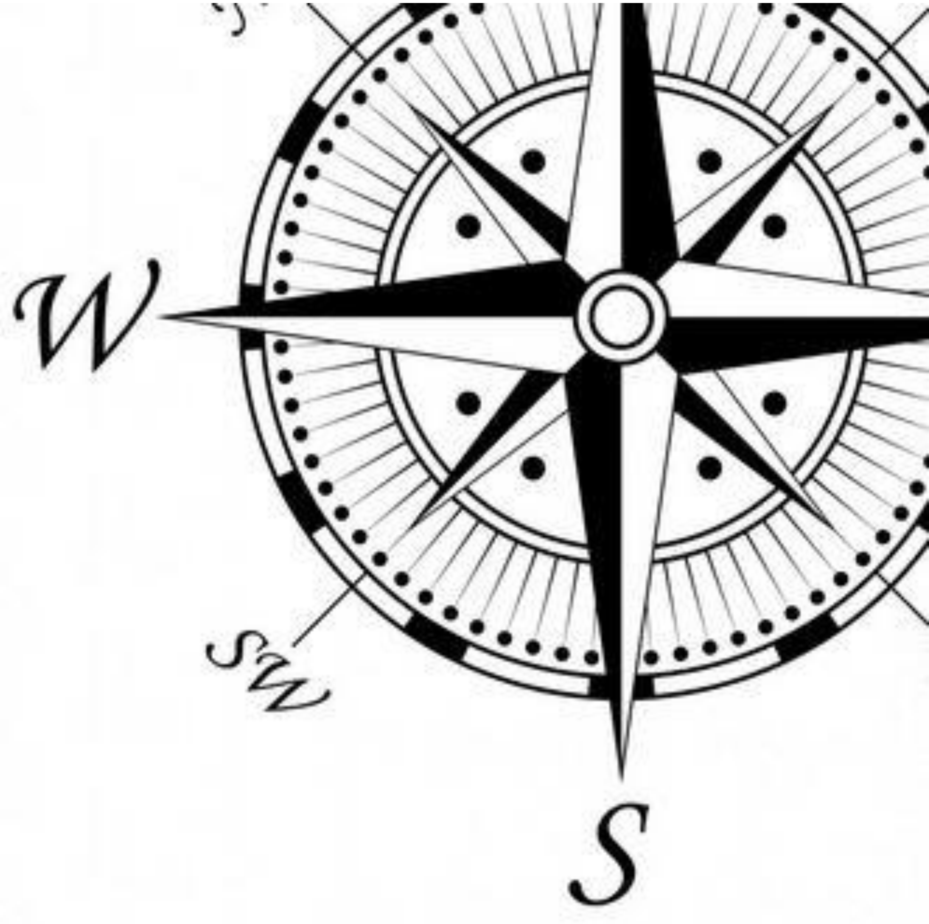
Learning Goals for this Chapter

- Understand the problem of vanishing gradients
  - And what can be done to solve it
- Understand and make use of LSTMs and GRUs
- Develop a baseline for a given problem statement
- Sucessful working with time series and sequential data
- Understand and deploy dropout with RNNs
- Pros and Cons of
  - Multilayer RNNs
  - Bidirectional RNNs

- Relevant chapters:
  - P6.3, S7 (2019) https://www.youtube.com/watch?v=QEw0qEa0E50

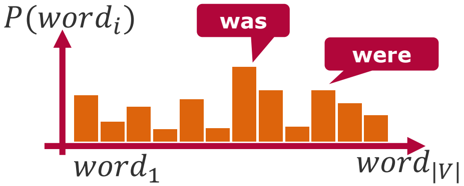
Topics Today

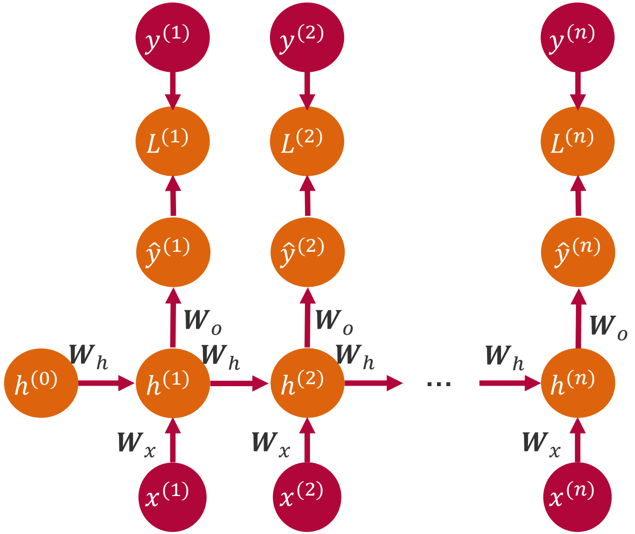
1. **Vanishing Gradients**
2. LSTM & GRU
3. Time Series Analysis
4. RNN Variations

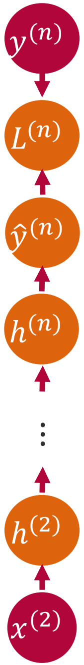


Language Model Dependencies

- Language model example:
  - „The mouse, which I saw yesterday, _____ here today as well.“
  - „The mice, which I saw yesterday, _____ here today as well.“



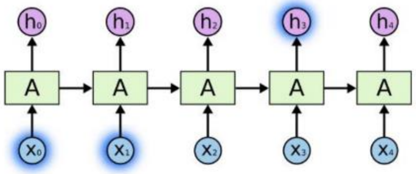




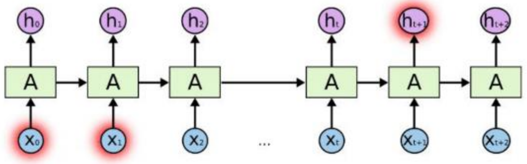
Long Range Dependencies

- Simple RNNs are „too simple“
  - Theoretically, they have at time step t access to information from far away past.
  - But, practically impossible to learn these **long range dependencies**.

Long-Term Dependency



- Short-term dependence:
  Bob is eating an **apple.**
- Long-term dependence:
  Context → **Bob** likes **apples**. He is hungry and decided to have a snack. So now he is eating an **apple.**



In theory, vanilla RNNs can handle arbitrarily long-term dependence.

In practice, it's difficult.

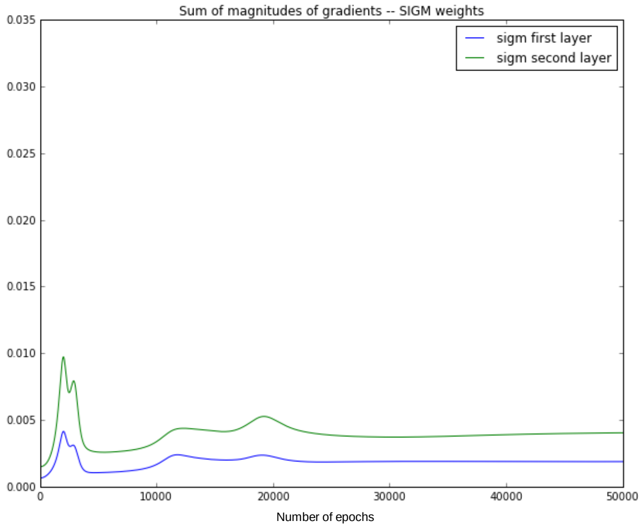
Vanishing Gradients

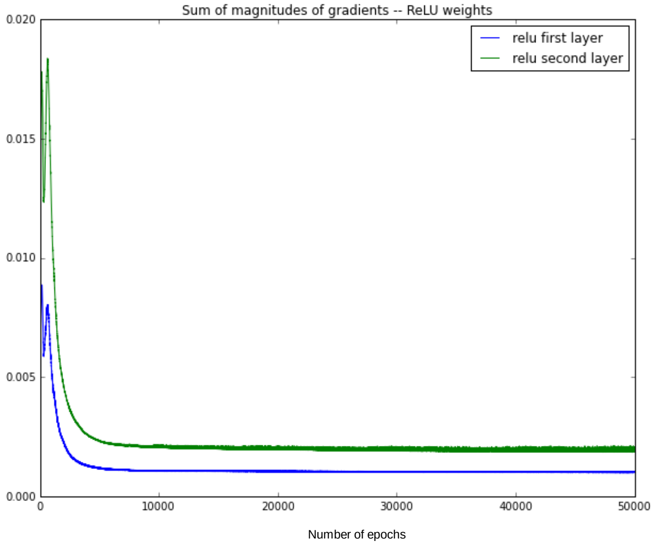
- Reason for poor RNN performance in practice for long range dependencies
  - **Vanishing gradient problem**
  - Can also be observed with deep feed forward networks
  - Each additional layer makes learning more difficult.
    - At some point, learning becomes impossible.
  - Gradients are getting smaller and smaller
    - At some point they get 0, i.e. they vanish
    - → Gradient descent can no longer „descent“

Yoshua Bengio, Patrice Simard, and Paolo Frasconi, “Learning Long-Term Dependencies with Gradient Descent Is Difficult”. In *IEEE Transactions on Neural Networks* 5, no. 2 (1994).

Vanishing Gradients: Effect

- Three-layer NN
  - Two hidden layers with 50 neurons each





https://cs224d.stanford.edu/notebooks/vanishing_grad_example.html

Exploding Gradients

- Closely related to vanishing gradients
  - Gradients can explode after repeated matrix multiplication
    - This is exactly what happens with backpropagation through time

- -> SGD update steps are too large (cf. Learning rate too high)

- Worst case: Inf or NaN results
  - -> restart training (from check point)

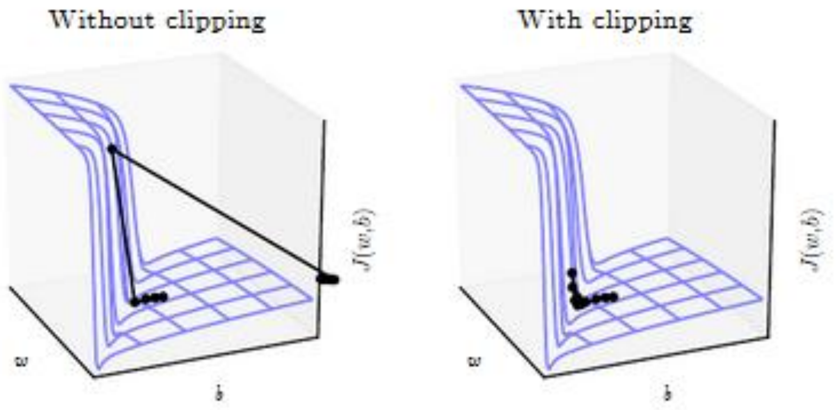
Exploding Gradients

- Simple hack
  - Clipping of gradients that are too large

$$\hat{\boldsymbol{g}} \leftarrow \frac{\partial L}{\partial \theta}$$

$$\textbf{if } \|\hat{\boldsymbol{g}}\| \geq \mathit{threshold} \textbf{ then}$$

$$\hat{\boldsymbol{g}} \leftarrow \frac{\mathit{threshold}}{\|\hat{\boldsymbol{g}}\|}\hat{\boldsymbol{g}} \textbf{ end if}$$



- Example:
  - Loss surface of an RNN with only two parameters: w and b
  - Steep walls in the landscape
  - Left: In cliff, large gradients -> large updates (bad!)
  - Right: large gradients are clipped -> direction of updates stays the same, but small step

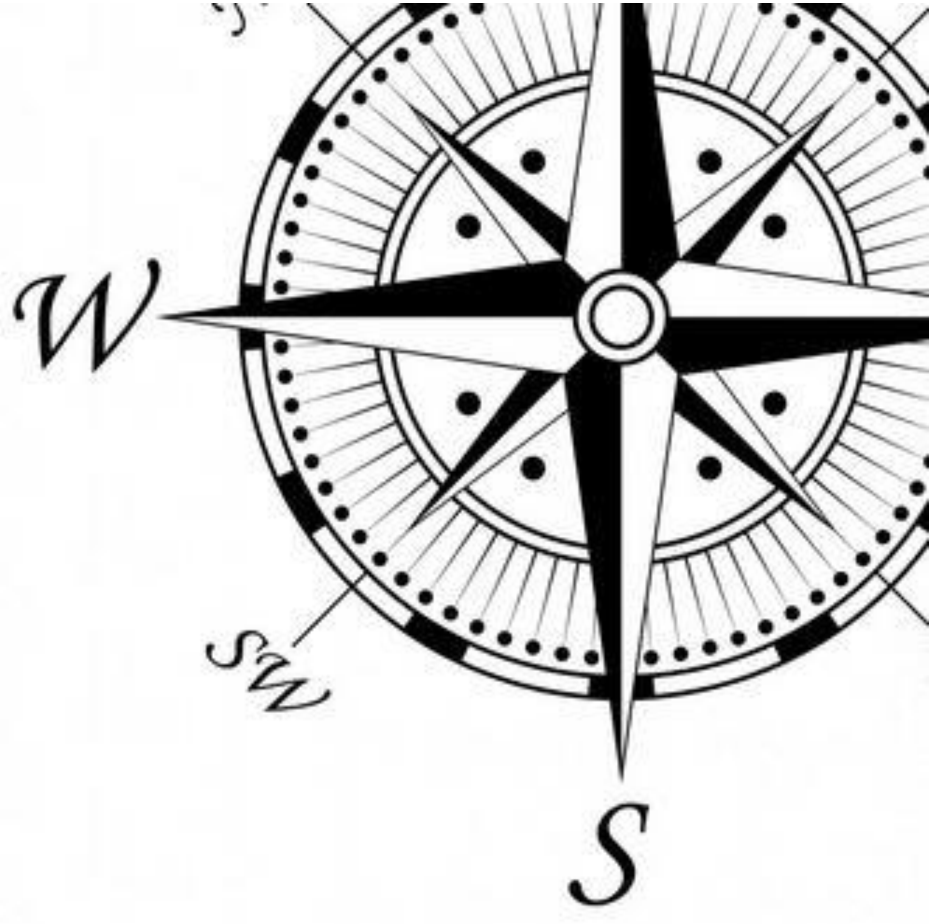
Vanishing Gradients

- Compute for an arbitrary RNN the gradients $\frac{\partial L^{(2)}}{\partial \boldsymbol{W}_h}$.
  - You can ignore all computations not involving the hidden layer weights.
    - Just assume random values.
  - Does it explode or vanish?

Start 5 4 3 2 1 End

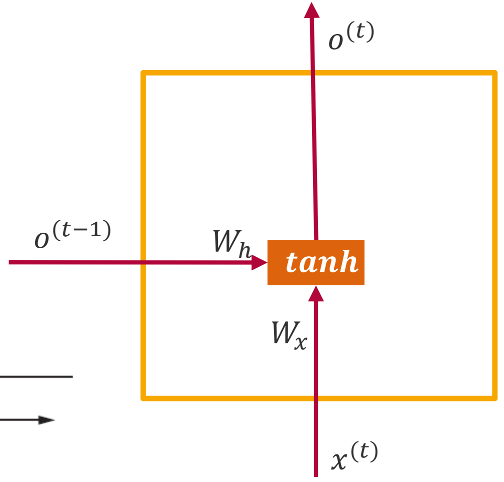
Topics Today

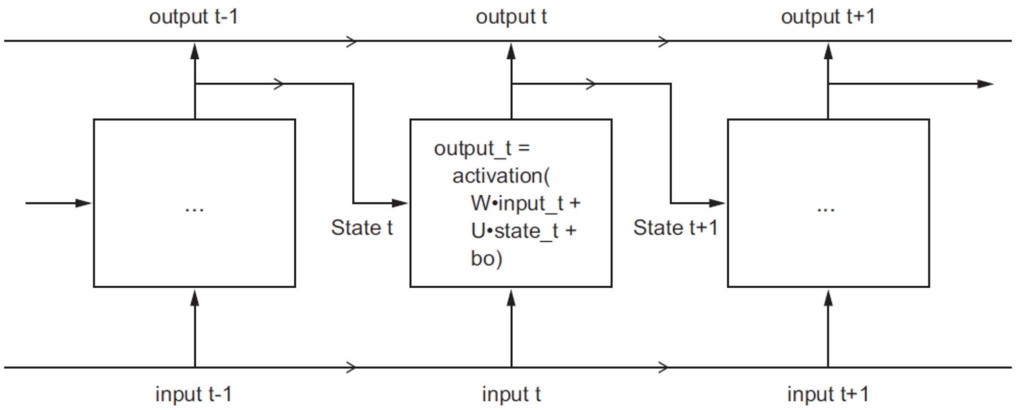
1. Vanishing Gradients
2. **LSTM & GRU**
3. Time Series Analysis
4. RNN Variations



Reminder: RNN Layer

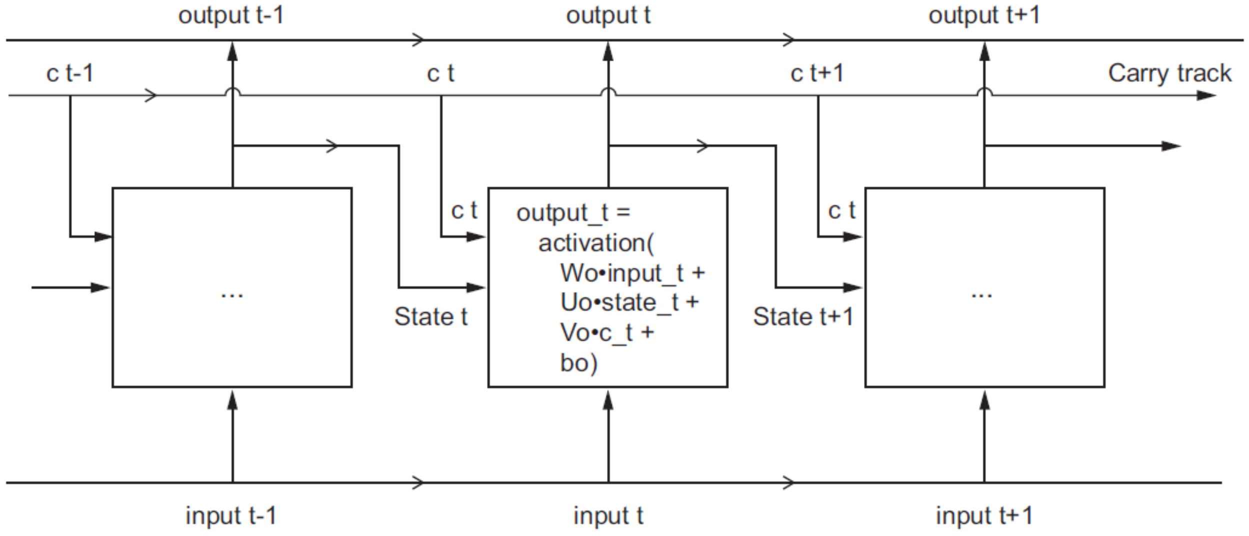
- Output of the layer at time step t
  - $h^{(t)} = \tanh(W_h h^{(t-1)} + W_x x^{(t)} + b_h)$



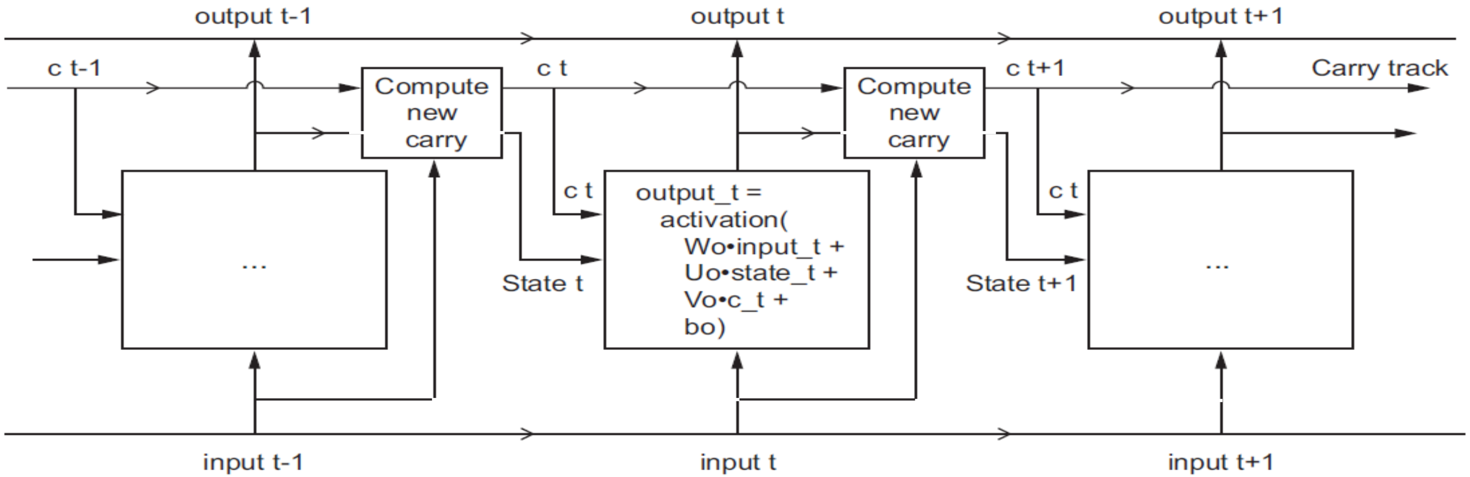


Long Short-Term Memory (LSTM)

- Idea: additional channel (c für carry), which allows access to information from the older past.
  - Old signals do not get smaller (-> no vanishing gradient).



Updating the Carry Signals



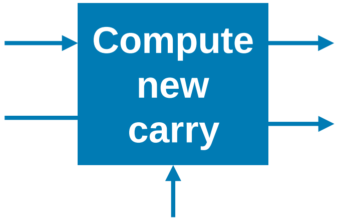
- How to compute the new carry signal?
  - Three distinct transformations
  - All three have the form of a SimpleRNN cell:
    - y = activation(dot(state_t, U) + dot(input_t, W) + b)
  - But all three transformations have their own weight matrices

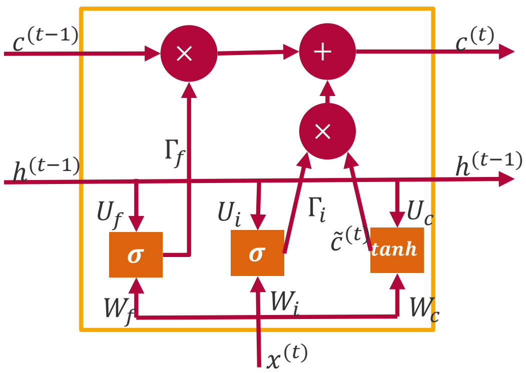
Long Short-Term Memory (LSTM)

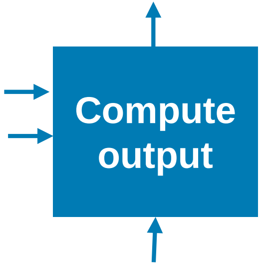
- Can also store old informtation from many time steps in the past
  - Can access this information at any time
  - Without the signals getting weaker (vanishing gradients)
- Implemented with **gates**
  - A **forget gate** can „forget“ information in the carry channel on purpose.
  - An **input gate** decides how much the carry signal gets updated.
  - An **output gate** decides how strong the influence of old information is.
- In general, the design of an RNN cell defines the hypothesis space.
  - What can be **modeled**?
  - Not: what is the cell **doing**.
- What a cell is doing depends on the **learned weights**.

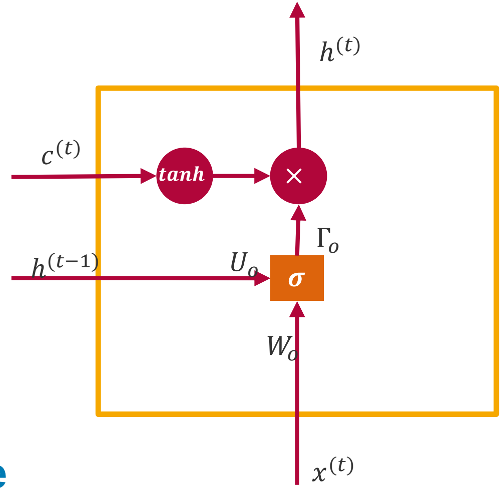
Sepp Hochreiter and Jürgen Schmidhuber, “Long Short-Term Memory,” *Neural Computation* 9, no. 8 (1997).

LSTM Computation







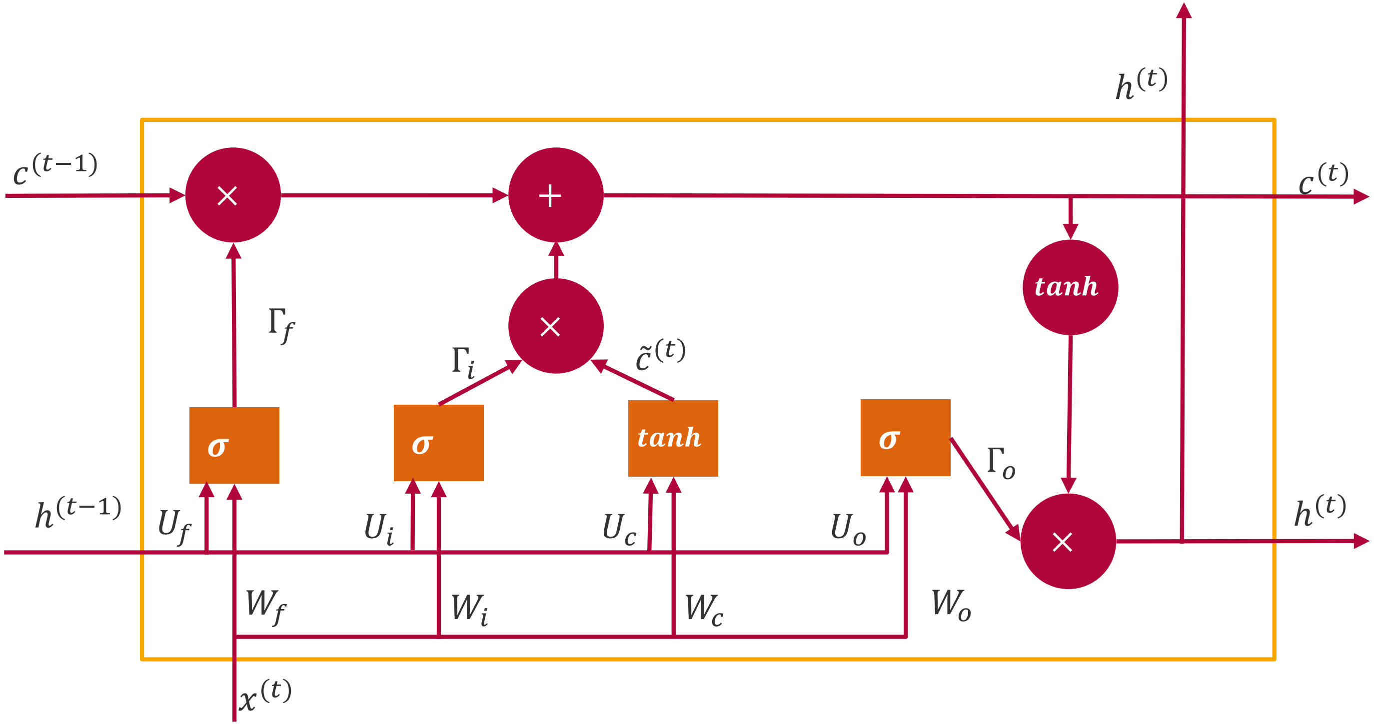


- New carry
  - $c^{(t)} = \Gamma_i * \tilde{c}^{(t)} + \Gamma_f * c^{(t-1)}$
- **Carry candidate**
  - $\tilde{c}^{(t)} = \tanh(U_c h^{(t-1)} + W_c x^{(t)} + b_c)$
- Output/Activation
  - $h^{(t)} = \Gamma_o * \tanh(c^{(t)})$

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

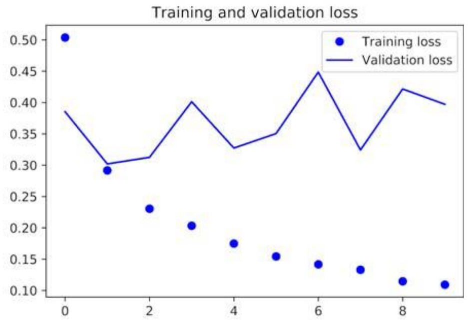
- Gates
  - **Input-Gate**
    - $\Gamma_i = \sigma(U_i h^{(t-1)} + W_i x^{(t)} + b_i)$
  - **Forget-Gate**
    - $\Gamma_f = \sigma(U_f h^{(t-1)} + W_f x^{(t)} + b_f)$
  - Output-Gate
    - $\Gamma_o = \sigma(U_o h^{(t-1)} + W_o x^{(t)} + b_o)$

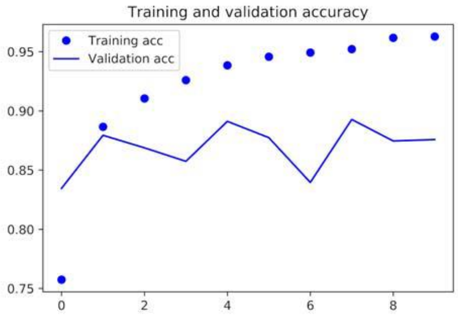
Graphical Representation of an LSTM Cell



Example LSTM Network

```
from keras.layers import LSTM
model = Sequential()
model.add(Embedding(max_features, 32))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop',
          loss='binary_crossentropy',
          metrics=['acc'])
history = model.fit(input_train, y_train,
                    epochs=10,batch_size=128,validation_split=0.2)
```





- 89% validation accuracy vs. 85% (simpleRNN) vs. 88% (dense feed-forward NN)
- Sentiment analysis not really the most stuited problem for LSTMs
  - Long range dependencies not so important
  - Bag-of-words approach sufficient
- More suitable problems: **question-answering**, **machine translation**

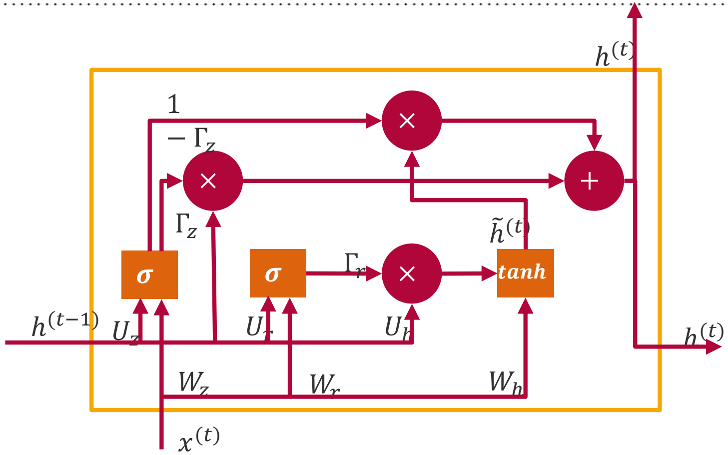
Gated Recurrent Unit (GRU)

- Simpler than LSTMs
  - **Reset gate**:
    - Decides how much information from the past should be forgotten
  - **Update gate**:
    - Decides how much information from the previous step should be forwarded to the next step
  - **Memory**
    - Stores relevant information from the past

- Gates can ignore parts of the memory
  - Extreme cases: ignore completely or copy completely

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

GRU Computation

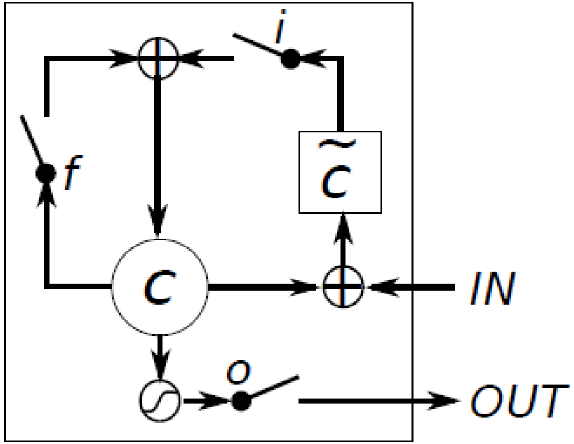


- Output/Activation
  - $h^{(t)} = \Gamma_z * h_{(t-1)} + (1 - \Gamma_z) * \tilde{h}^{(t)}$
- Memory
  - $\tilde{h}^{(t)} = \tanh(\Gamma_r * U_h h^{(t-1)} + W_h x^{(t)})$

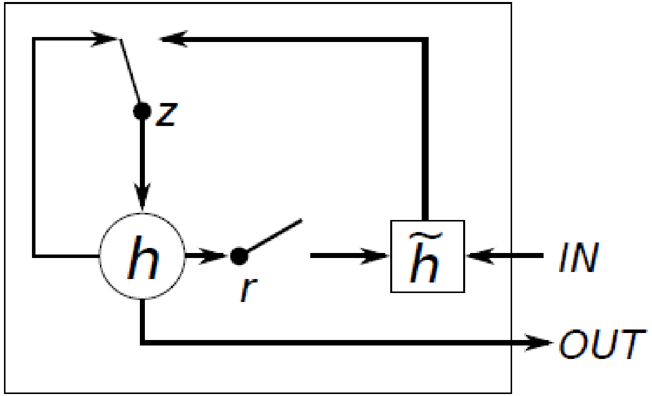
https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be

- Gates
  - Update-Gate
    - $\Gamma_z = \sigma(U_z h^{(t-1)} + W_z x^{(t)} + b_z)$
  - Reset-Gate
    - $\Gamma_r = \sigma(U_r h^{(t-1)} + W_r x^{(t)} + b_r)$

LSTM vs. GRU



LSTM



GRU

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.

LSTM & GRU

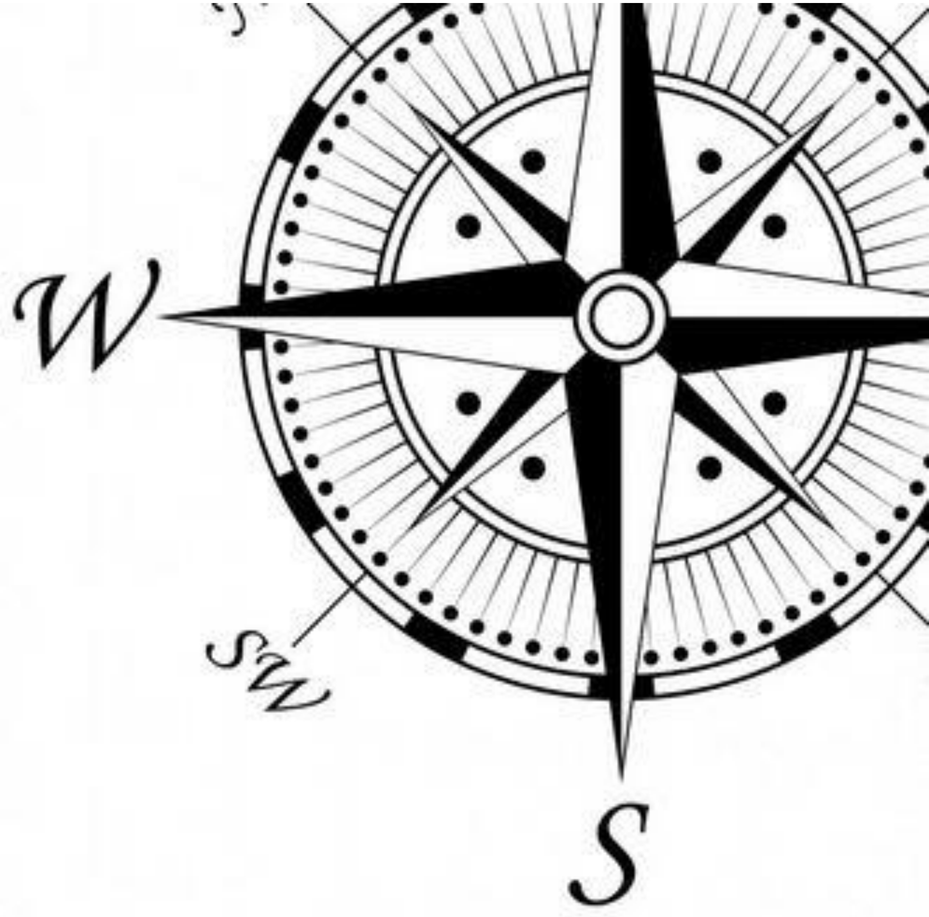
- Given the following input text: „The cars, that I see, are red.“
  - Conceptually, go through an RNN with an LSTM or GRU layer which reads one word per time step. How can the word „are“ (vs. „is“) be correctly predicted?
- Which is better: LSTM or GRU?
  - Speed
  - Number of parameters
  - Tasks
  - Complexity

- E.g. *Empirical Evaluation of Gated RNNs on Sequence Modeling*
  - https://arxiv.org/pdf/1412.3555v1.pdf

Start 5 4 3 2 1 End

Topics Today

1. Vanishing Gradients
2. LSTM & GRU
3. **Time Series Analysis**
4. RNN Variations

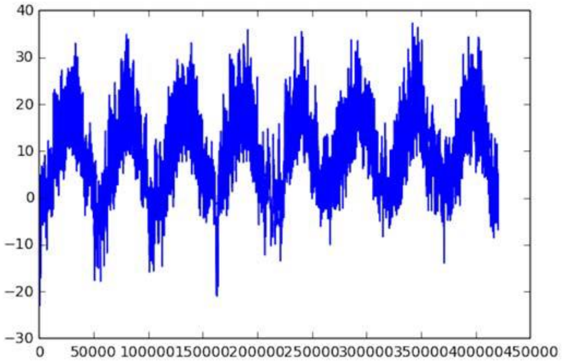


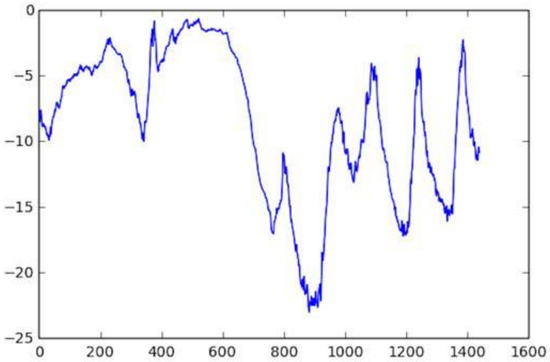
Time Series Analysis: Reading Input Data

```
wget https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip
unzip jena_climate_2009_2016.csv.zip
import os
import numpy as np
data_dir = '/users/krestel/Downloads/jena_climate'
fname = os.path.join(data_dir, 'jena_climate_2009_2016.csv')
f = open(fname)
data = f.read()
f.close()
lines = data.split('\n')
header = lines[0].split(',')
lines = lines[1:]
print(header)
print(len(lines))
float_data = np.zeros((len(lines), len(header) - 1))
for i, line in enumerate(lines):
        values = [float(x) for x in line.split(',')[1:]]
        float_data[i, :] = values
```

Predicting Temperature

```
from matplotlib import pyplot as plt
temp = float_data[:, 1] <1> temperature (in degrees Celsius)
plt.plot(range(len(temp)), temp)
plt.plot(range(1440), temp[:1440])
```





- Normalization

```
mean = float_data[:200000].mean(axis=0)
float_data -= mean
std = float_data[:200000].std(axis=0)
float_data /= std
```

Data Preparation

```
def generator(data,lookback,delay,min_index,max_index,shuffle=False,batch_size=128,step=6):
    if max_index is None:
        max_index = len(data) - delay - 1
    i = min_index + lookback
    while 1:
        if shuffle:
            rows = np.random.randint(min_index + lookback, max_index, Size=batch_size)
        else:
          if i + batch_size >= max_index:
              i = min_index + lookback
          rows = np.arange(i, min(i + batch_size, max_index))
          i += len(rows)
        samples = np.zeros((len(rows),lookback // step, data.shape[-1]))
        targets = np.zeros((len(rows),))
        for j, row in enumerate(rows):
          indices = range(rows[j] - lookback, rows[j], step)
          samples[j] = data[indices]
          targets[j] = data[rows[j] + delay][1]
        yield samples, targets
```

Data Preprocessing

```
lookback = 1440
step = 6
delay = 144
batch_size = 128
train_gen = generator(float_data,
        lookback=lookback,
        delay=delay,
        min_index=0,
        max_index=200000,
        shuffle=True,
        step=step,
        batch_size=batch_size)
val_gen = generator(float_data, lookback=lookback, delay=delay, min_index=200001,
max_index=300000, step=step, batch_size=batch_size)
test_gen = generator(float_data, lookback=lookback, delay=delay, min_index=300001,
max_index=None, step=step, batch_size=batch_size)
val_steps = (300000 - 200001 - lookback)
test_steps = (len(float_data) - 300001 - lookback)
```

Simple Baseline

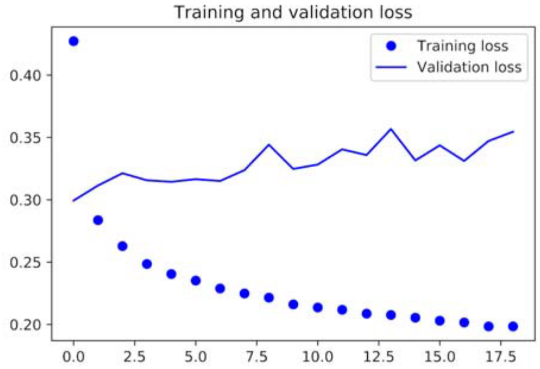
- Predicting the temperature in 24 hours: no changes!

```
def evaluate_naive_method():
    batch_maes = []
    for step in range(val_steps):
        samples, targets = next(val_gen)
        preds = samples[:, -1, 1]
        mae = np.mean(np.abs(preds - targets))
        batch_maes.append(mae)
    print(np.mean(batch_maes))
evaluate_naive_method()
celsius_mae = 0.29 * std[1]
```

2.57°C

NN Baseline

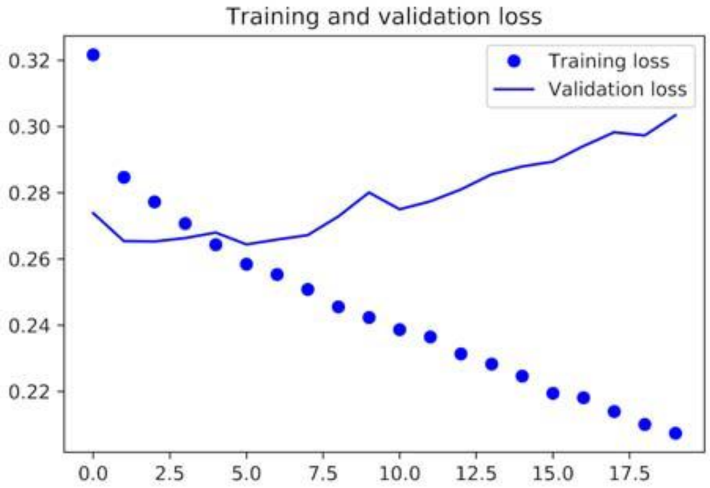
```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop
model = Sequential()
model.add(layers.Flatten(input_shape=(lookback // step, float_data.shape[-1])))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(1))
model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen, steps_per_epoch=500,
epochs=20, validation_data=val_gen, validation_steps=val_steps)
import matplotlib.pyplot as plt
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```



NN Baseline

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop
model = Sequential()
model.add(layers.GRU(32, input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))
model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=20,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

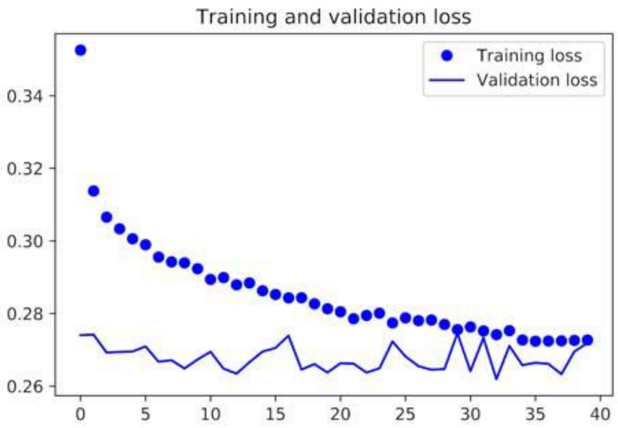
- MAE of 0.265 right before overfitting starts
  - Corresponds to an average error of 2.35° C
- Prevent overfitting: **dropout**
  - Randomly zeros out input units of a layer in order to break happenstance correlations in the training data that the layer is exposed to.



Recurrent Dropout to Prevent Overfitting

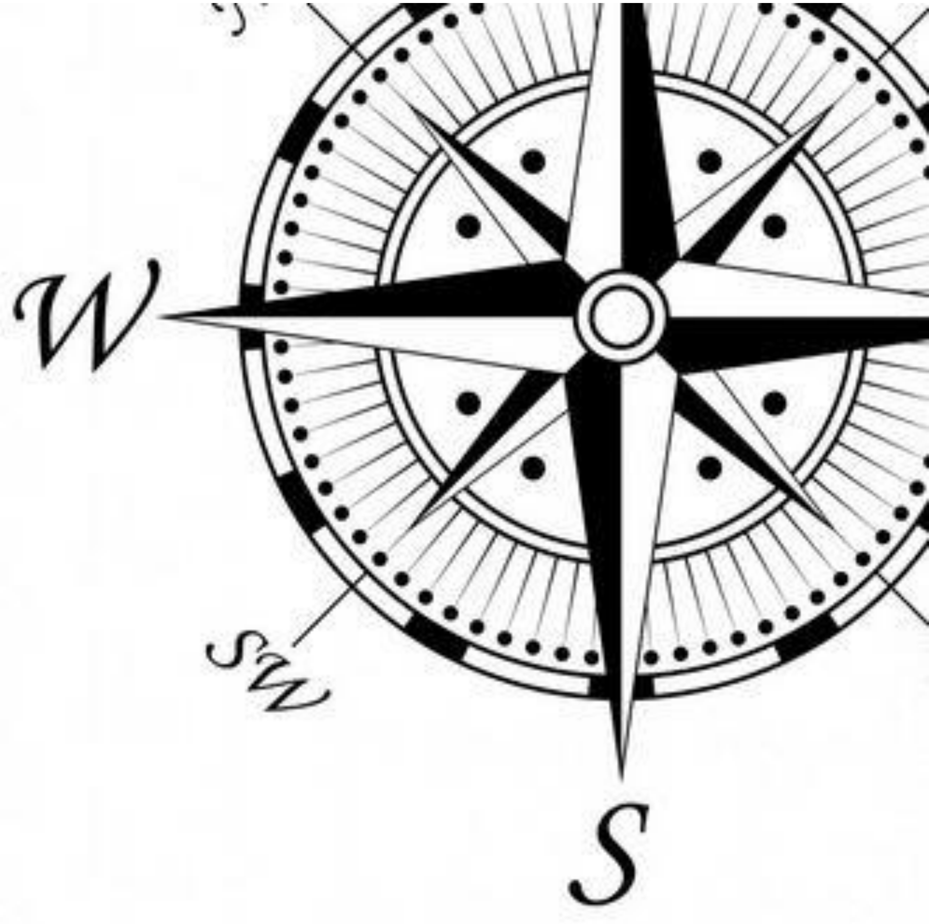
- The same dropout mask should be used **for all time steps**
  - Otherwise more damage than good, since learning becomes impossible
- Not only a dropout mask **for the input**, but also **for the activation**
  - Again, same for each time step

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop
model = Sequential()
model.add(layers.GRU(32, dropout=0.2, recurrent_dropout=0.2,
                     input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))
model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=40,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```



Topics Today

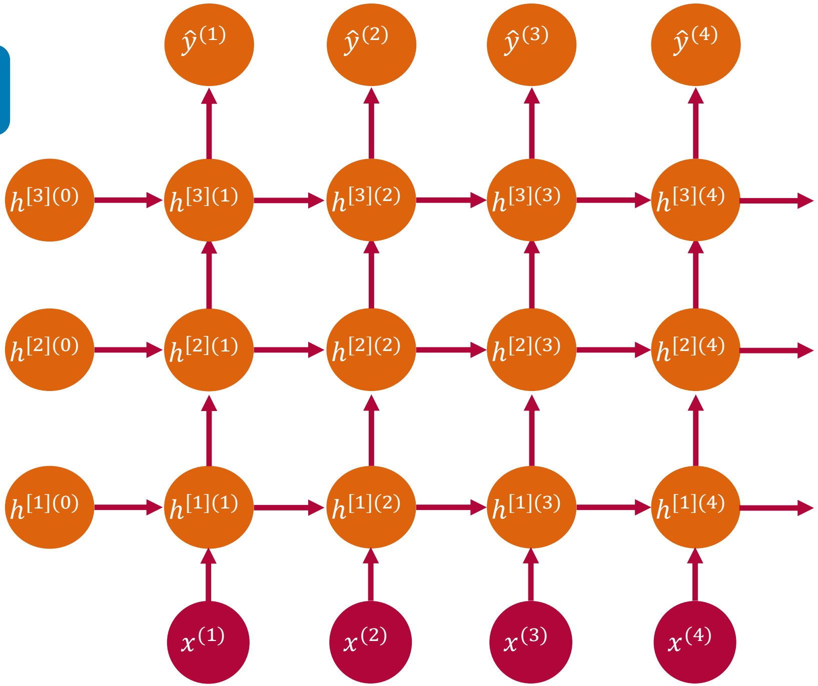
1. Vanishing Gradients
2. LSTM & GRU
3. Time Series Analysis
4. **RNN Variations**



Stacked RNNs

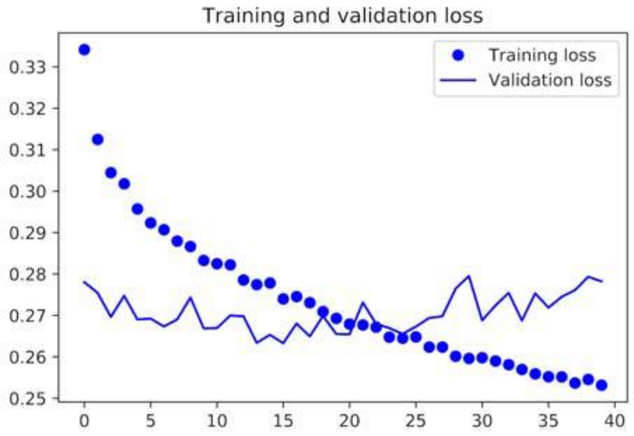
- Increasing the capacity of the network
  - More units per layer
  - More layers
- RNNs are already "deep" in one dimension (time)
- We can also make them "deep" in another dimension
  - Multi-layer or stacked RNN
- This allows the network to compute more complex representations
  - The lower RNNs should compute lower-level features and the higher RNNs should compute higher-level features.

Google Translate: 7 large LSTM layers



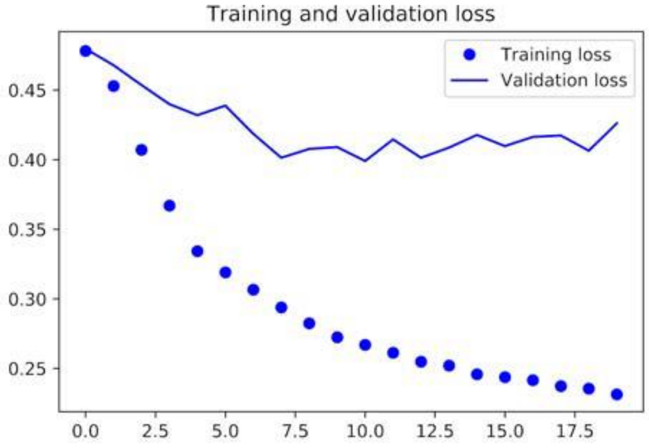
Stacked RNNs: Example

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop
model = Sequential()
model.add(layers.GRU(32,
         dropout=0.1,
         recurrent_dropout=0.5,
         return_sequences=True,
         input_shape=(None, float_data.shape[-1])))
model.add(layers.GRU(64, activation='relu',
         dropout=0.1,
         recurrent_dropout=0.5))
         model.add(layers.Dense(1))
model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
         steps_per_epoch=500,
         epochs=40,
         validation_data=val_gen,
         validation_steps=val_steps)
```



Switched Order: Temperature

```
def generator(data, lookback, delay, min_index,
                 max_index, shuffle=False,
                 batch_size=128, step=6):
    if max_index is None:
        max_index = len(data) - delay - 1
    i = min_index + lookback
    while 1:
        if shuffle:
            rows = np.random.randint(min_index + lookback,
                          max_index, Size=batch_size)
        else:
          if i + batch_size >= max_index:
              i = min_index + lookback
          rows = np.arange(i, min(i + batch_size, max_index
          i += len(rows)
        samples = np.zeros((len(rows),lookback // step, data.shape[-1]))
        targets = np.zeros((len(rows),))
        for j, row in enumerate(rows):
          indices = range(rows[j] - lookback, rows[j], step)
          samples[j] = data[indices]
          targets[j] = data[rows[j] + delay][1]
        yield samples, targets
        yield samples[:, ::-1, :], targets
```



Switched Order: IMDB

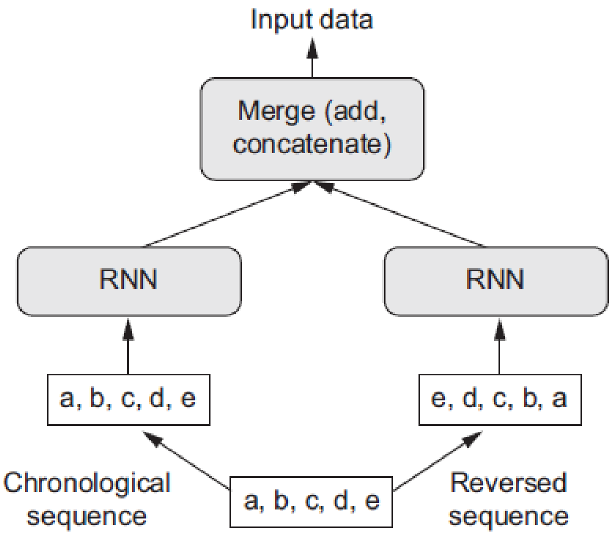
```
from keras.datasets import imdb
from keras.preprocessing import sequence
from keras import layers
from keras.models import Sequential
max_features = 10000
maxlen = 500
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
x_train = [x[::-1] for x in x_train]
x_test = [x[::-1] for x in x_test]
x_train = sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)
model = Sequential()
model.add(layers.Embedding(max_features, 128))
model.add(layers.LSTM(32))
model.add(layers.Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop',
        loss='binary_crossentropy',
        metrics=['acc'])
history = model.fit(x_train, y_train,
        epochs=10,
        batch_size=128,
        validation_split=0.2)
```

Swiched order

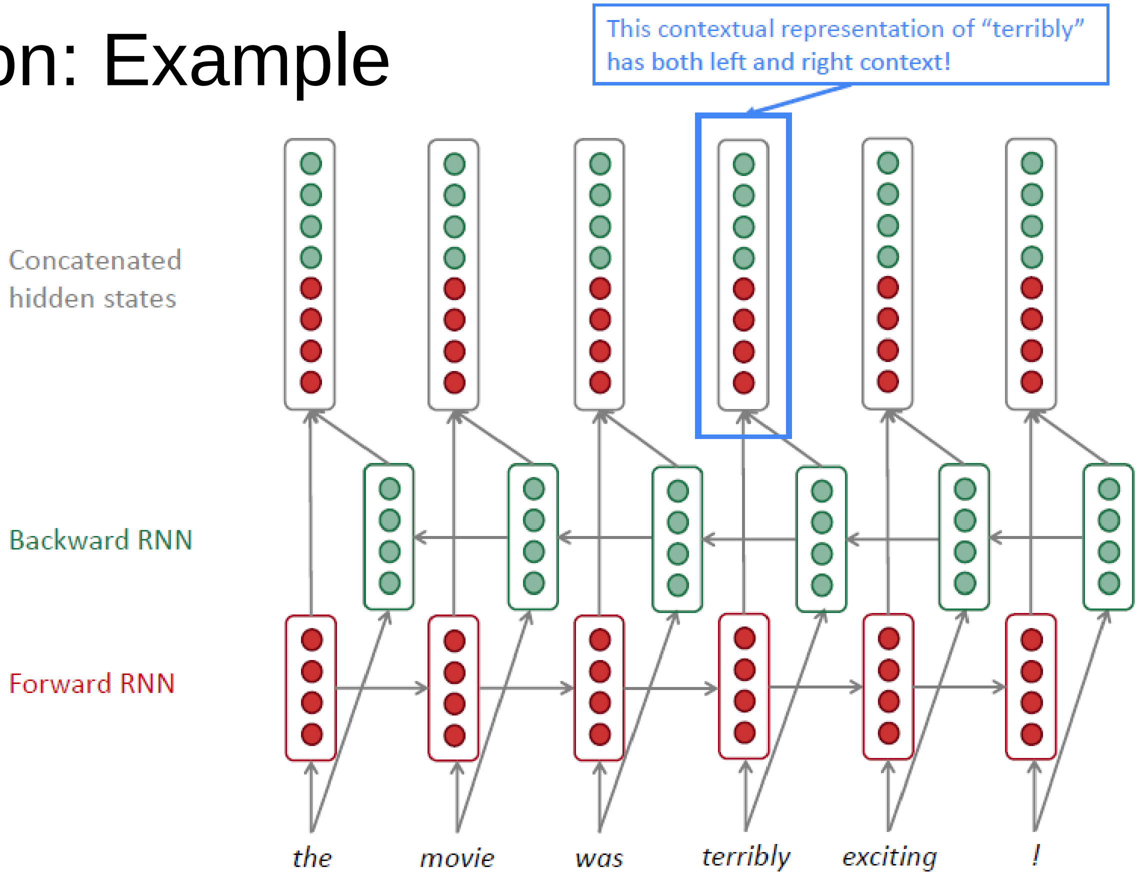
No performance gain, but also no decrease

Bi-directional RNNs

- Learned representation is different if input ordering is changed
- Meaningful but different information are very interesting for machine learning
  – Combination mostly superior
  – Different aspects of the data
  – Combine Strenghts, couterbalance weaknesses
  – Ensemble methods
- Bidirectional RNNs combine both input orderings
  – Only applicable when you have access to the entire input sequence.
  – They are **not** applicable to language modeling as the future tokens are not accessible.
- If you do have entire input sequence, use bi-directional encoding by default.



Review Classification: Example



Bi-directional LSTM: IMDB

```
model = Sequential()
model.add(layers.Embedding(max_features, 32))
model.add(layers.Bidirectional(layers.LSTM(32)))
model.add(layers.Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop',
          loss='binary_crossentropy',
          metrics=['acc'])
history = model.fit(x_train, y_train,
          epochs=10,
          batch_size=128,
          validation_split=0.2)
```

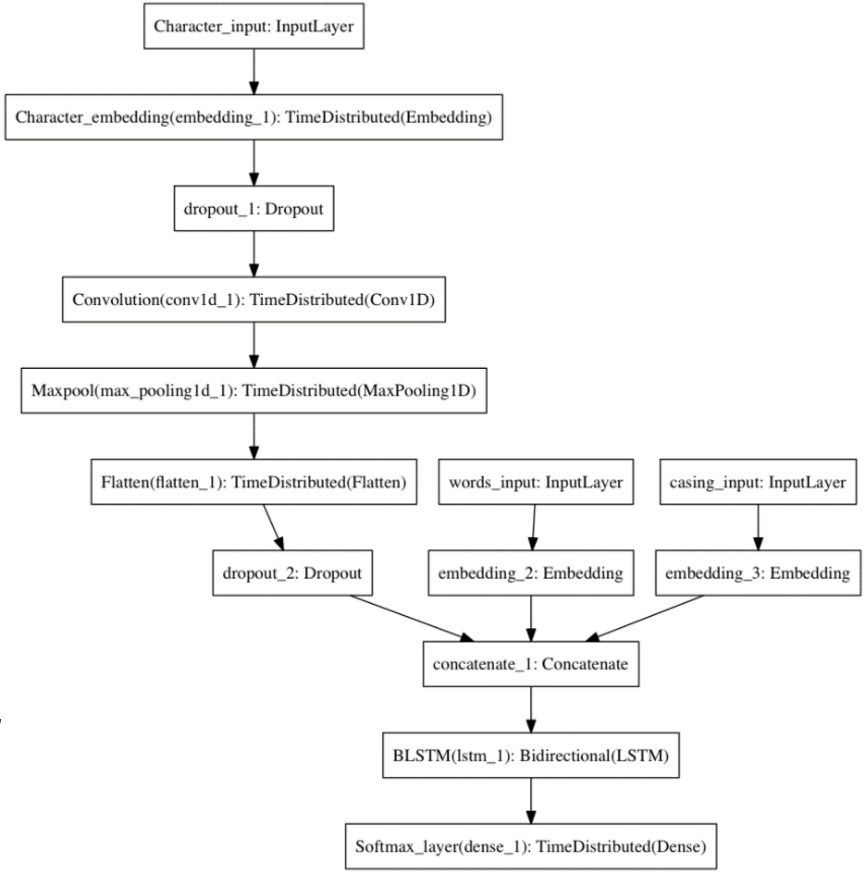
- 89% accuracy

Bi-directional GRU: Temperature

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop
model = Sequential()
model.add(layers.Bidirectional(layers.GRU(32),
        input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))
model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
        steps_per_epoch=500,
        epochs=40,
        validation_data=val_gen,
        validation_steps=val_steps)
```

- Approximately as good as a regular GRU layer!

Combining LSTM & CNN



Chiu, J. P., & Nichols, E. (2016). Named entity recognition with bidirectional LSTM-CNNs. *Transactions of the association for computational linguistics*, *4*, 357-370. https://arxiv.org/pdf/1511.08308

Learning Goals for this Chapter

- Understand the problem of vanishing gradients
  - And what can be done to solve it
- Understand and make use of LSTMs and GRUs
- Develop a baseline for a given problem statement
- Sucessful working with time series and sequential data
- Understand and deploy dropout with RNNs
- Pros and Cons of
  - Multilayer RNNs
  - Bidirectional RNNs

- Relevant chapters:
  - P6.3, S7 (2019) https://www.youtube.com/watch?v=QEw0qEa0E50