Deep Learning in NLP

Horacio Rodríguez

Outline

From linear models to NN

classifier

$$\text{classify}(\boldsymbol{x}, \boldsymbol{\theta}) = \underset{\boldsymbol{y} \in \mathcal{L}}{\text{argmax}}\ \text{score}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta})$$

linear classifier

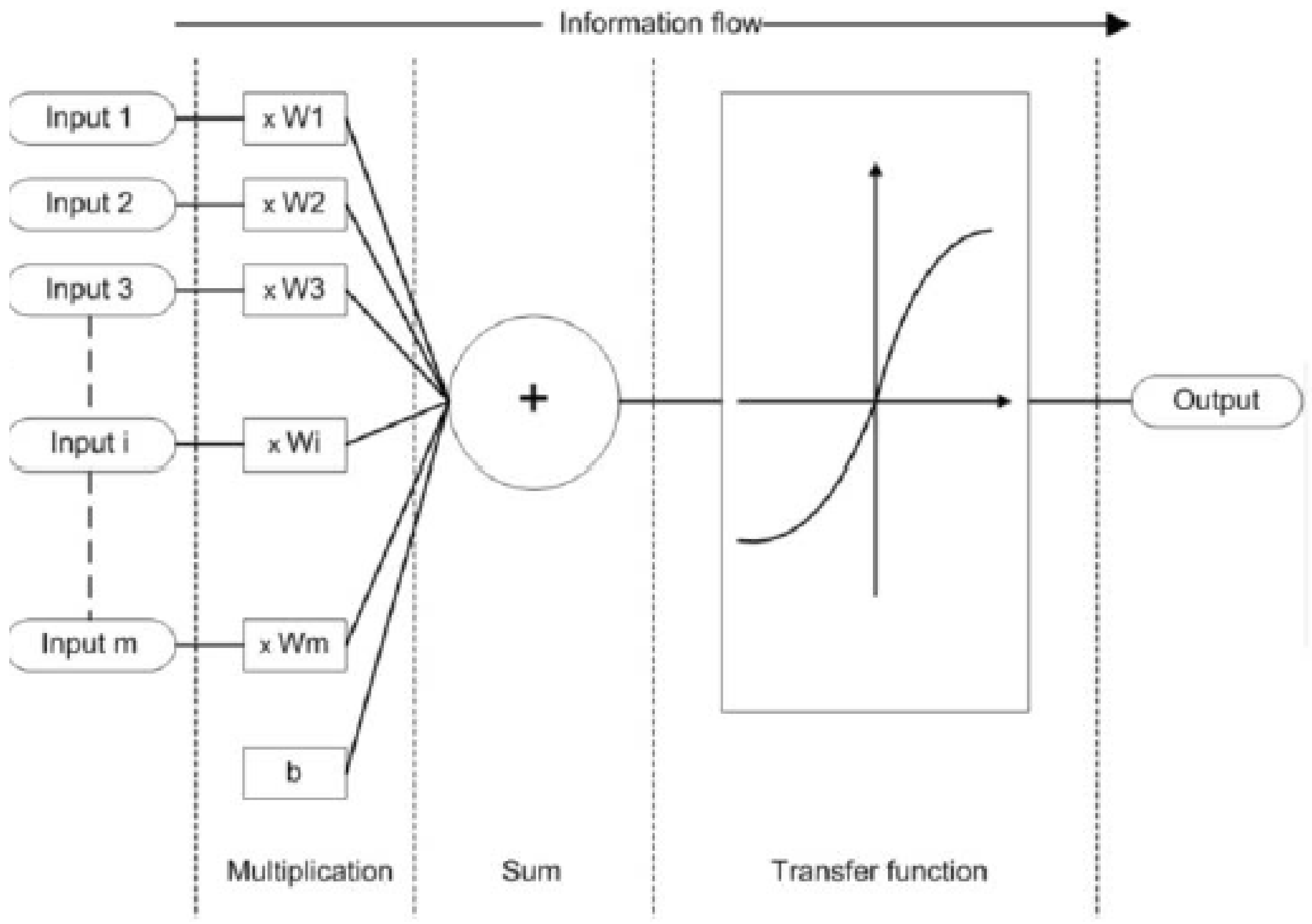
$$\text{score}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\theta}) = \sum_i \theta_i f_i(\boldsymbol{x}, \boldsymbol{y})$$

neural layer = affine transform + nonlinearity

$$\boldsymbol{z}^{(1)} = \underset{\text{nonlinearity}}{g}\left(\underbrace{W^{(0)}\boldsymbol{x} + \boldsymbol{b}^{(0)}}_{\text{affine transform}}\right)$$

From linear models to NN

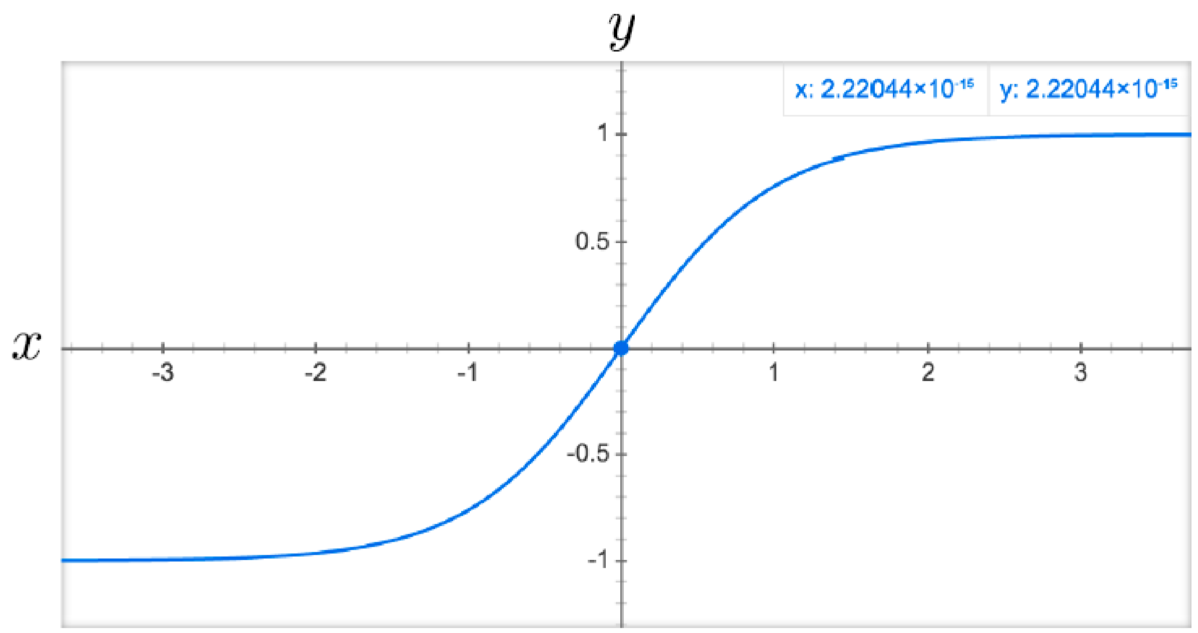
The scheme of a neuron's operation, input data (input) is multiplied by weights (W), summed, added bias (b) and the result is sent to the input of some activation function



From linear models to NN

Common nonlinearities

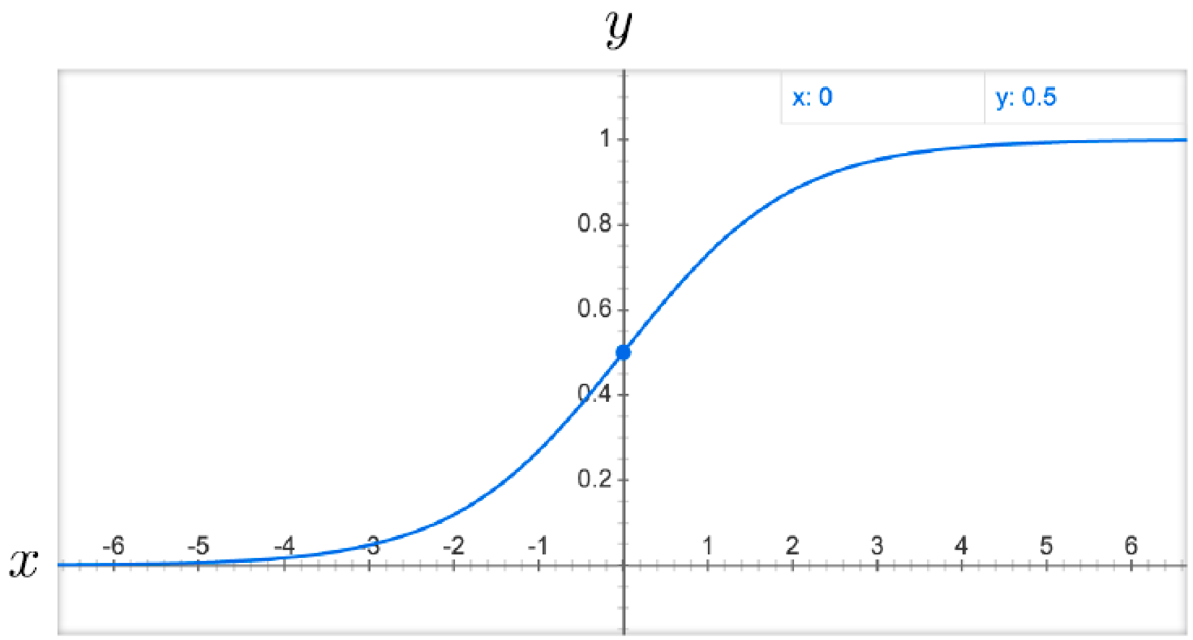
tanh: $y = \tanh(x)$



From linear models to NN

Common nonlinearities

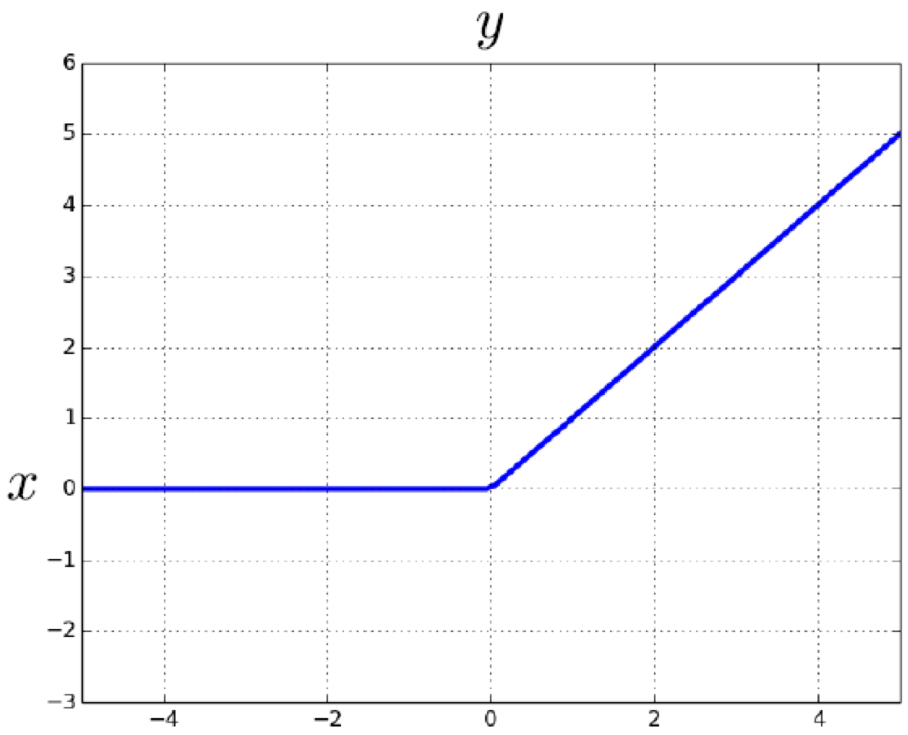
(logistic) sigmoid: $y = \dfrac{1}{1 + \exp\{-x\}}$



From linear models to NN

Common nonlinearities

rectified linear unit (ReLU): $y = \max(0, x)$



14

NN models for NLP

- Loss Functions (for training the model)
  - Hinge (binary)

$$L_{hinge(binary)}(\hat{y}, y) = \max(0, 1 - y \cdot \hat{y})$$

  - Hinge (multiclass)

$$L_{hinge(multiclass)}(\hat{\mathbf{y}}, \mathbf{y}) = \max(0, 1 - (\hat{y}_t - \hat{y}_k))$$

  - Log loss

$$L_{log}(\hat{\mathbf{y}}, \mathbf{y}) = \log(1 + exp(-(\hat{y}_t - \hat{y}_k))$$

  - Categorical cross-entropy loss

$$L_{cross-entropy}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_i y_i \log(\hat{y}_i)$$

  - Ranking losses

$$L_{ranking(margin)}(\mathbf{x}, \mathbf{x}') = \max(0, 1 - (NN(\mathbf{x}) - NN(\mathbf{x}')))$$

NN models for NLP

- Output transformations
  - SoftMax (most popular for classification)

$$\mathbf{x} = x_1, \ldots, x_k$$

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}}$$

NN models for NLP

- Learning

Algorithm 1 Online Stochastic Gradient Descent Training

```
1: Input: Function $f(\mathbf{x};\theta)$ parameterized with parameters $\theta$.
2: Input: Training set of inputs $\mathbf{x_1},\ldots,\mathbf{x_n}$ and outputs $\mathbf{y_1},\ldots,\mathbf{y_n}$.
3: Input: Loss function $L$.
4: while stopping criteria not met do
5:     Sample a training example $\mathbf{x_i},\mathbf{y_i}$
6:     Compute the loss $L(f(\mathbf{x_i};\theta),\mathbf{y_i})$
7:     $\hat{\mathbf{g}} \leftarrow$ gradients of $L(f(\mathbf{x_i};\theta),\mathbf{y_i})$ w.r.t $\theta$
8:     $\theta \leftarrow \theta + \eta_k \hat{\mathbf{g}}$
9: return $\theta$
```

NN models for NLP

- Learning
  - SGD is the most popular training algorithm
  - But there are others:
    - Stochastic gradient descent with momentum remembers the update weights at each iteration, and determines the next update as a convex combination of the gradient and the previous update
    - Nesterov accelerated gradient (Nesterov Momentum)
    - Adagrad
    - RMSProp and Adadelta
    - Adam - adaptive moment estimation
    - Adamax
    - Resilient propagation (Rprop)

NN models

- Software
  - Scikit-learn is not enough
  - Several Python-based model libraries
  - Low level: Theano, CNN, pyCNN
  - Midle level: TensorFlow, Chainer, Torch
  - High leval: Keras, PENNE, Lasagne, pyLearn2, Caffe

Logistic regression with Keras

```
import keras
print keras.__version__
from keras import models
from keras import layers
from keras.models import Input, Model
from keras.layers import Dense, Dropout
```

```
def buildKerasLogRegModel(parameters, x_train, y_train, x_test, y_test, epochs):
    inputs = Input(shape=(parameters['numFeatures'],))
    output = Dense(1, activation='sigmoid')(inputs)
    kerasLogRegModel = Model(inputs, output)
    kerasLogRegModel.compile(optimizer='sgd',
                                  loss = 'binary_crossentropy', metrics=['accuracy'])
    kerasLogRegModel.optimizer.lr = 0.001
    kerasLogRegModel.fit(x=x_train, y=y_train, epochs = epochs,
                             validation_data = (x_test, y_test),verbose=0)
    return kerasLogRegModel
```

Logistic regression with Keras

```
def predKerasLogRegModel(kerasLogRegModel, x_test, y_test):
    y_pred = kerasLogRegModel.predict(x_test)
    return y_pred > 0.5
```

```
kerasSimpleFNNModel = buildKerasLogRegModel(
                    parameters, x_train, y_train, x_test, y_test, epochs)
y_pred = predKerasSimpleFNNModel(
                    kerasSimpleFNNModel, x_test, y_test)
p, r, f1, a, m = evaluateClassifier(y_pred, y_test)
```

NN models for NLP

- Feed-forward neural Networks (FFNM)
- Convolutional NN (CNN)
- Recurrent NN (RNN)
- Autoencoders
- Recursive NN (RecNN)
- Gated models
  - Long Short Time Models (LSTM)
  - Gated Recurrent Units (GRU)

NN models for NLP

- Feed-forward neural Networks (FFNM)
- Convolutional NN (CNN)
- Recurrent NN (RNN)
- Autoencoders
- Recursive NN (RecNN)
- Gated models
  - Long Short Time Models (LSTM)
  - Gated Recurrent Units (GRU)
- Attention-based models

FFNN

- feed-forward neural Network (FFNN)
  - general structure for an NLP classification system
    - 1 Extract a set of core linguistic features f_1, …, f_k that are relevant for predicting the output class.
    - 2. For each feature f_i of interest, retrieve the corresponding vector $v(f_i)$.
    - 3. Combine the vectors (either by concatenation, summation or a combination of both into an input vector x.
    - 4. Feed x into a non-linear classifier (feed-forward neural network).

FFNN

- Feed-forward neural Network
  - Simple Perceptron

$$NN_{Perceptron}(\mathbf{x}) = \mathbf{x}\mathbf{W} + \mathbf{b}$$

$$\mathbf{x} \in \mathbb{R}^{d_{in}}, \quad \mathbf{W} \in \mathbb{R}^{d_{in} \times d_{out}}, \quad \mathbf{b} \in \mathbb{R}^{d_{out}}$$

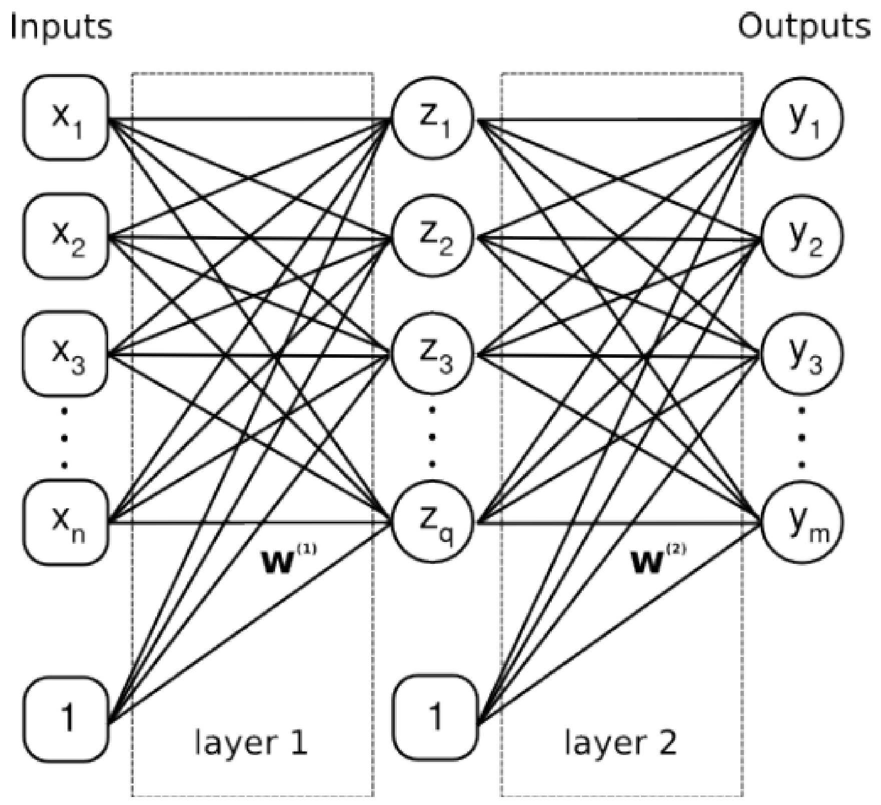
  - 1-layer Multi Layer Perceptron (MLP1)

$$NN_{MLP1}(\mathbf{x}) = g(\mathbf{x}\mathbf{W}^{\mathbf{1}} + \mathbf{b}^{\mathbf{1}})\mathbf{W}^{\mathbf{2}} + \mathbf{b}^{\mathbf{2}}$$

$$\mathbf{x} \in \mathbb{R}^{d_{in}}, \quad \mathbf{W}^{\mathbf{1}} \in \mathbb{R}^{d_{in} \times d_1}, \quad \mathbf{b}^{\mathbf{1}} \in \mathbb{R}^{d_1}, \quad \mathbf{W}^{\mathbf{2}} \in \mathbb{R}^{d_1 \times d_2}, \quad \mathbf{b}^{\mathbf{2}} \in \mathbb{R}^{d_2}$$

FFNN

- Simple Feed Forward NN
  - Example of neuron feed-forward fully connected network with one hidden layer



FFNN

- Simple Feed Forward NN
  - Example of neuron feed-forward fully connected network with one hidden layer

$$y_m = f^{(2)}\left(\sum_{q=0}^{Q} w_{mq}^{(2)} f^{(1)}\left(\sum_{i=0}^{N} w_{qn}^{(1)} x_n\right)\right)$$

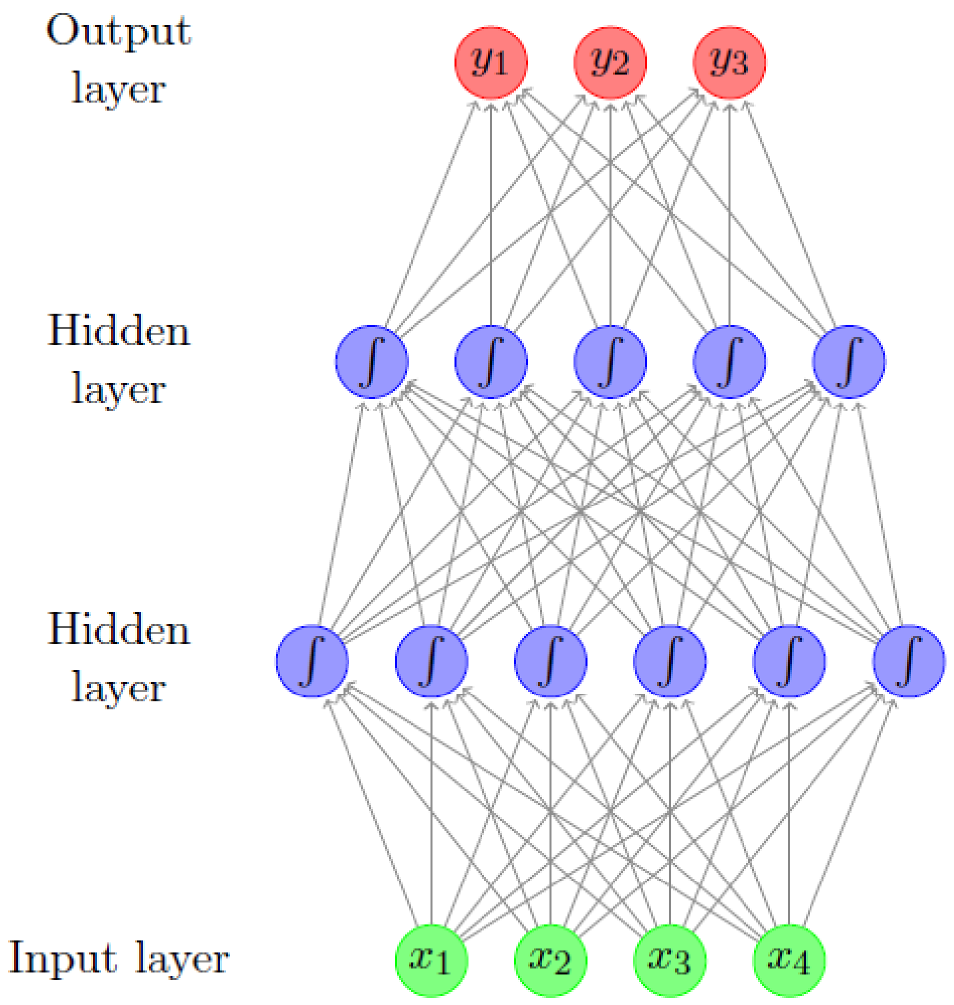
FFNN

- Feed-forward neural Network
  - 2-layer Multi Layer Perceptron (MLP2)

$$\mathrm{NN}_{\mathrm{MLP2}}(\mathbf{x}) = (g^2(g^1(\mathbf{x}\mathbf{W}^1 + \mathbf{b}^1)\mathbf{W}^2 + \mathbf{b}^2))\mathbf{W}^3$$

$$\begin{aligned}
\mathrm{NN}_{\mathrm{MLP2}}(\mathbf{x}) &= \mathbf{y} \\
\mathbf{h}^1 &= g^1(\mathbf{x}\mathbf{W}^1 + \mathbf{b}^1) \\
\mathbf{h}^2 &= g^2(\mathbf{h}^1\mathbf{W}^2 + \mathbf{b}^2) \\
\mathbf{y} &= \mathbf{h}^2\mathbf{W}^3
\end{aligned}$$

2 hidden layers FFNN



FFNN with Keras

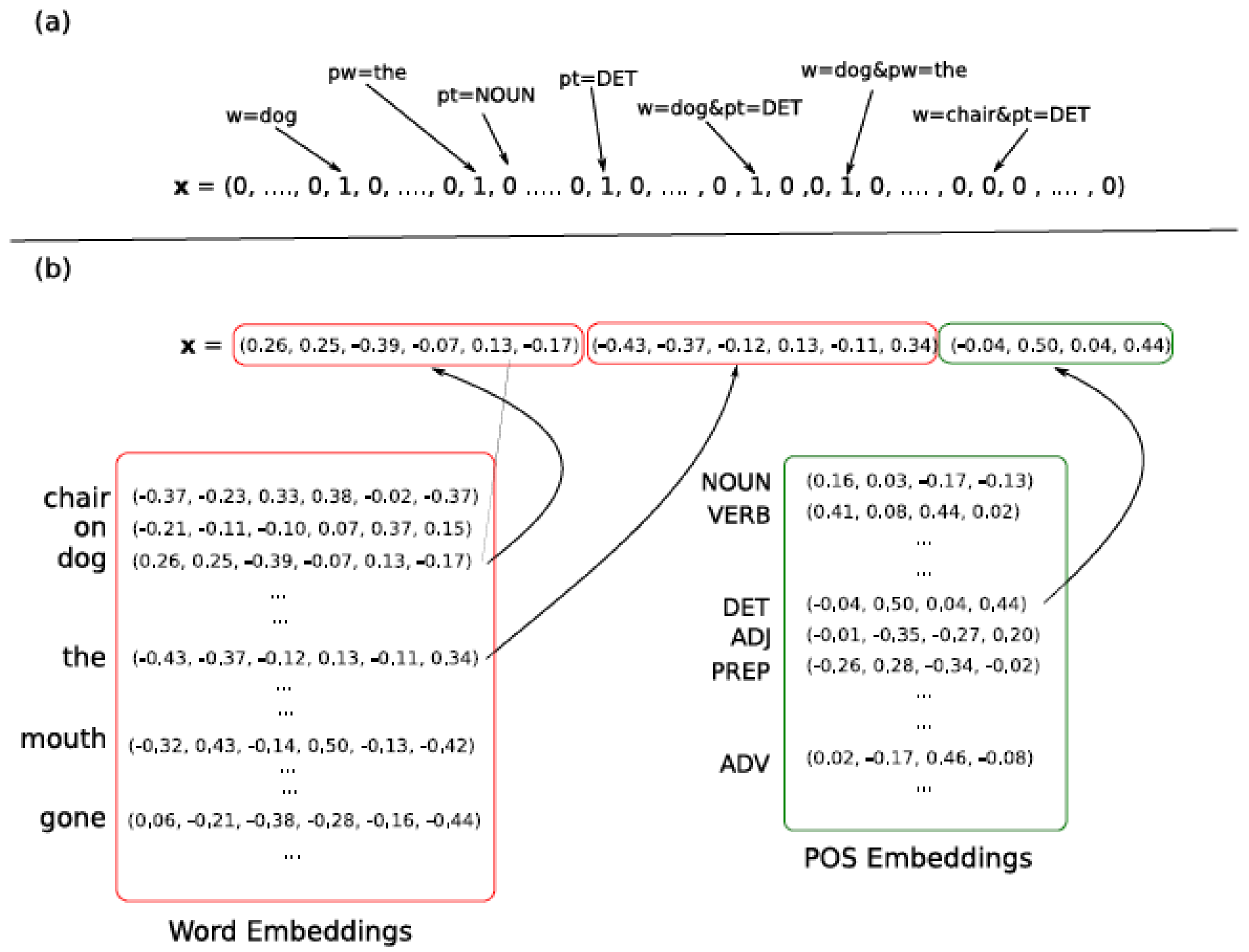
```
def buildKerasDeepFNNModel(parameters, x_train, y_train, x_test, y_test, epochs, numLayers):
    inputs = Input(shape=(parameters['numFeatures'],))
    X = Dense(parameters['numFeatures']/10, activation='relu')(inputs)
    X = Dropout(0.4)(X)
    for layer in range(numLayers -1):
        X = Dense(parameters['numFeatures']/20, activation='relu')(inputs)
        X = Dropout(0.3)(X)
    output = Dense(1, activation='sigmoid')(X)
    kerasDeepFNNModel = Model(inputs, output)
    kerasDeepFNNModel.compile(
                        optimizer='adam', loss = 'binary_crossentropy', metrics=['accuracy'])
    kerasDeepFNNModel.fit(
                        x=x_train, y=y_train, epochs = epochs,
                        validation_data = (x_test, y_test),verbose=0)
    return kerasDeepFNNModel
```

NN models for NLP

- Sparse vs. dense feature representations.
  - Two encodings of the information:
    - current word is \dog"; previous word is \the"; previous pos-tag is \DET".
    - (a) Sparse feature vector. Each dimension represents a feature. Feature combinations receive their own dimensions. Feature values are binary. Dimensionality is very high.
    - (b) Dense, embeddings-based feature vector. Each core feature is represented as a vector. Each feature corresponds to several input vector entries. No explicit encoding of feature combinations. Dimensionality is low. The feature-to-vector mappings come from an embedding table.

NN models for NLP

- Sparse vs. dense feature representations.



(a)
w=dog
pw=the
pt=NOUN
pt=DET
w=dog&pt=DET
w=dog&pw=the
w=chair&pt=DET
x = (0, ..., 0, 1, 0, ..., 0, 1, 0 ..... 0, 1, 0, ... , 0 , 1, 0 ,0, 1, 0, ... , 0, 0, 0 , ... , 0)
(b)
x = (0.26, 0.25, -0.39, -0.07, 0.13, -0.17) (-0.43, -0.37, -0.12, 0.13, -0.11, 0.34) (-0.04, 0.50, 0.04, 0.44)
chair (-0.37, -0.23, 0.33, 0.38, -0.02, -0.37)
on (-0.21, -0.11, -0.10, 0.07, 0.37, 0.15)
dog (0.26, 0.25, -0.39, -0.07, 0.13, -0.17)
the (-0.43, -0.37, -0.12, 0.13, -0.11, 0.34)
mouth (-0.32, 0.43, -0.14, 0.50, -0.13, -0.42)
gone (0.06, -0.21, -0.38, -0.28, -0.16, -0.44)
Word Embeddings
NOUN (0.16, 0.03, -0.17, -0.13)
VERB (0.41, 0.08, 0.44, 0.02)
DET (-0.04, 0.50, 0.04, 0.44)
ADJ (-0.01, -0.35, -0.27, 0.20)
PREP (-0.26, 0.28, -0.34, -0.02)
ADV (0.02, -0.17, 0.46, -0.08)
POS Embeddings

NN models for NLP

- Encoders
  - a function to represent a word sequence as a vector
    - Simplest: average word embeddings:

$$f_{avg}(\boldsymbol{x}) = \frac{1}{|\boldsymbol{x}|} \sum_{i=1}^{|\boldsymbol{x}|} emb(x_i)$$

NN models for NLP

- Embedding Layers
  - c(.) is a function from core features to an input vector.
  - It is common for c to extract the embedding vector associated with each feature, and concatenate them

$$
\begin{aligned}
\mathbf{x} = c(f_1, f_2, f_3) &= [v(f_1); v(f_2); v(f_3)] \\
NN_{MLP1}(\mathbf{x}) &= NN_{MLP1}(c(f_1, f_2, f_3)) \\
&= NN_{MLP1}([v(f_1); v(f_2); v(f_3)]) \\
&= (g([v(f_1); v(f_2); v(f_3)]\mathbf{W^1} + \mathbf{b^1}))\mathbf{W^2} + \mathbf{b^2}
\end{aligned}
$$

NN models for NLP

- Embedding Layers
  - Another common choice for c is to sum the embedding vectors (this assumes the embedding vectors all share the same dimensionality)

$$\begin{aligned}\mathbf{x} = c(f_1, f_2, f_3) &= v(f_1) + v(f_2) + v(f_3) \\ NN_{MLP1}(\mathbf{x}) &= NN_{MLP1}(c(f_1, f_2, f_3)) \\ &= NN_{MLP1}(v(f_1) + v(f_2) + v(f_3)) \\ &= (g((v(f_1) + v(f_2) + v(f_3))\mathbf{W}^1 + \mathbf{b}^1))\mathbf{W}^2 + \mathbf{b}^2\end{aligned}$$

NN models for NLP

- Embedding Layers
  - Sometimes embeddings v(fi) result from an “embedding layer" or “lookup layer". Consider a vocabulary of |V| words, each embedded as a d dimensional vector. The collection of vectors can then be thought of as a |V| x d embedding matrix E in which each row corresponds to an embedded feature.

$$v(f_i) = \mathbf{f_i E}$$

$$CBOW(f_1, ..., f_k) = \sum_{i=1}^{k} (\mathbf{f_i E}) = (\sum_{i=1}^{k} \mathbf{f_i})\mathbf{E}$$

NN models for NLP

- Feed-forward neural Networks (FFNM)
- Convolutional NN (CNN)
- Recurrent NN (RNN)
- Autoencoders
- Recursive NN (RecNN)
- Gated models
  - Long Short Time Models (LSTM)
  - Gated Recurrent Units (GRU)
- Attention-based models

CNN

- Basic Convolution & Pooling
  - Predictions based on ordered sets of items (e.g. the sequence of words in a sentence, the sequence of sentences in a document and so on).
  - Apply a non-linear (learned) function over each instantiation of a k-word sliding window over the sentence. This function (also called filter) transforms a window of k words into a d dimensional vector that captures important properties of the words.
  - A CNN is designed to identify indicative local predictors in a large structure, and combine them to produce a fixed size vector representation of the structure, capturing these local aspects that are most informative for the prediction task at hand.

CNN

- Convolutional NN
  - Srihari proposes Three Mechanisms of Convolutional Neural Networks
    - Local Receptive Fields
    - Subsampling
    - Weight Sharing
  - Instead of treating input to a fully connected network two layers are used:
    - Layer of convolution
    - Layer of subsampling (pooling)

CNN

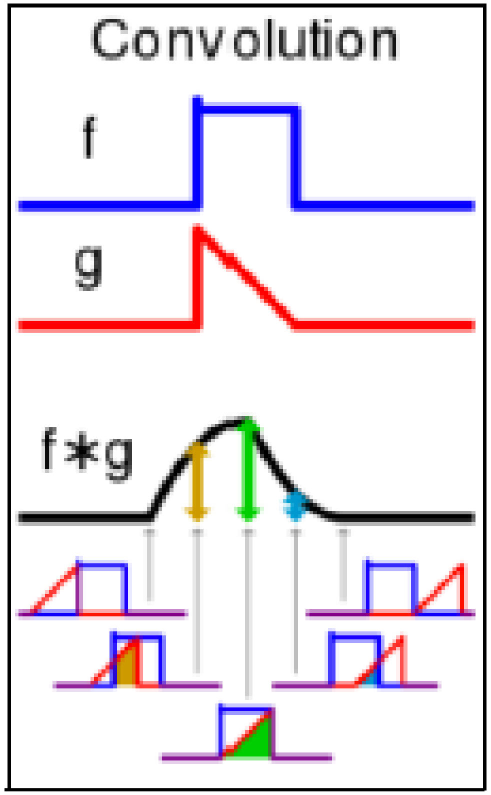
- Convolution
  - Input *f*(*t*) is convolved with a kernel *g*(*t*), sometimes named filter

$$\boxed{(f*g)(t) \equiv \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\,d\tau}$$

  - Express each function in terms of a dummy variable τ
  - Reflex g: g(τ) $\rightarrow$ g(-τ)
  - Add an offset t for allow g to slide along τ axis.
  - Integrate the product of f and g

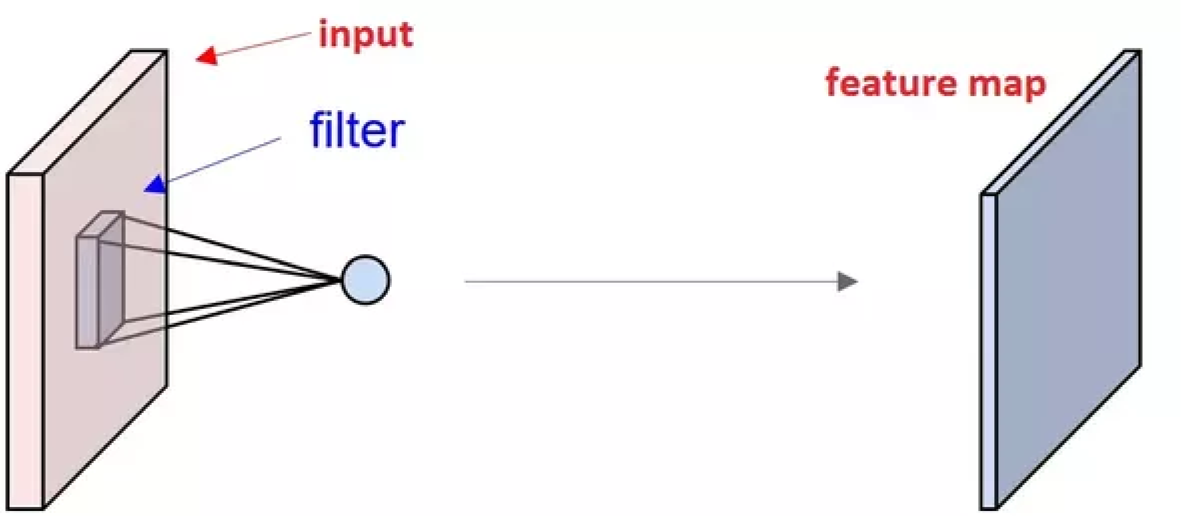
CNN

- Convolution



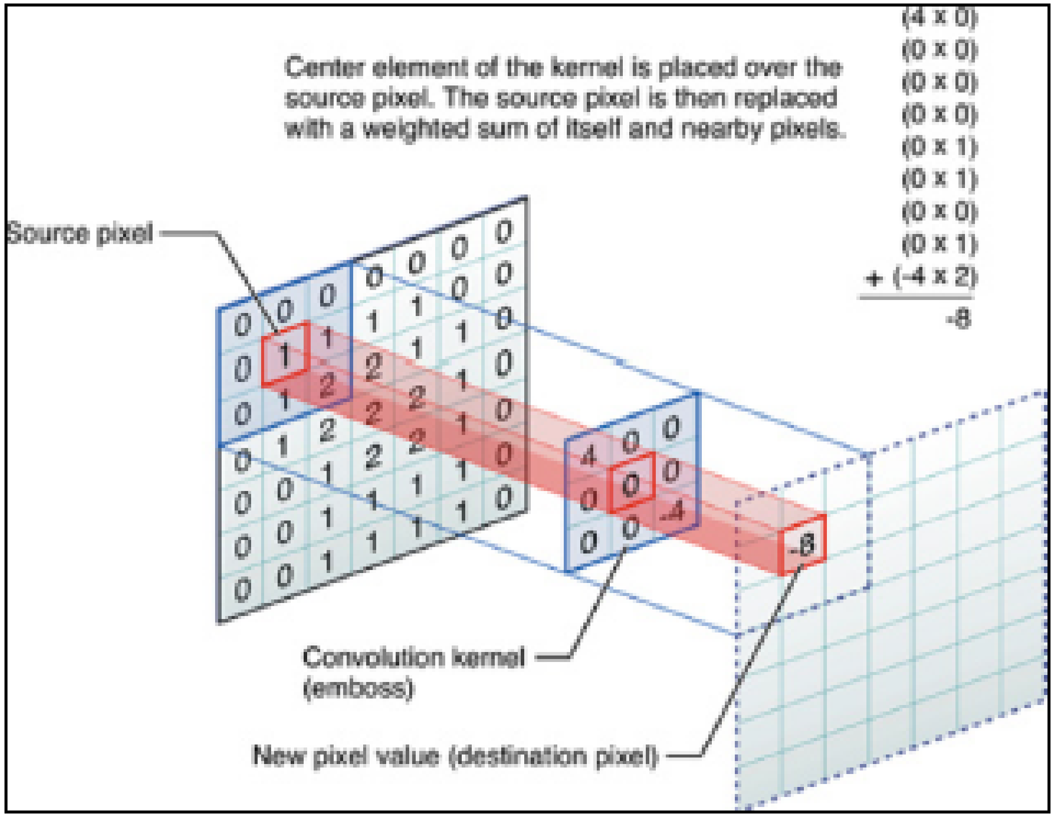
CNN

- Convolution
  - The feature map is the output of one filter applied to the previous layer. A given filter is drawn across the entire previous layer, moved one token at a time. Each position results in an activation of the neuron and the output is collected in the feature map.



CNN

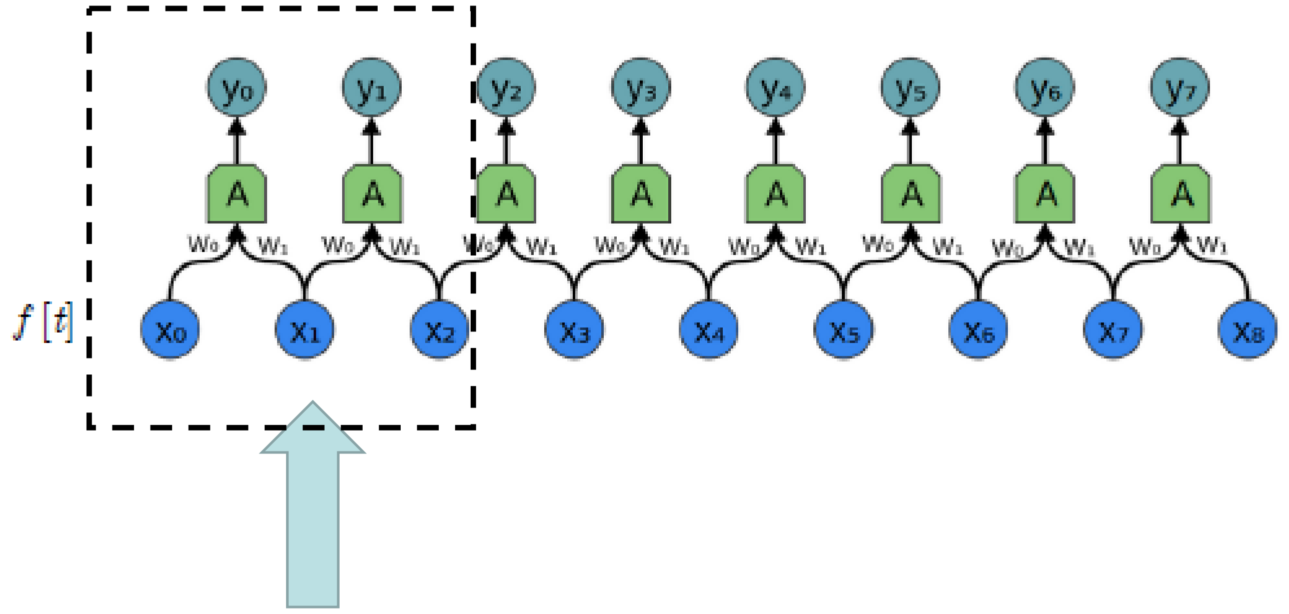
- Convolution in 2D



Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)

CNN

- Convolution in 1D
  - 2-gram model



Kernel $g(t)$:

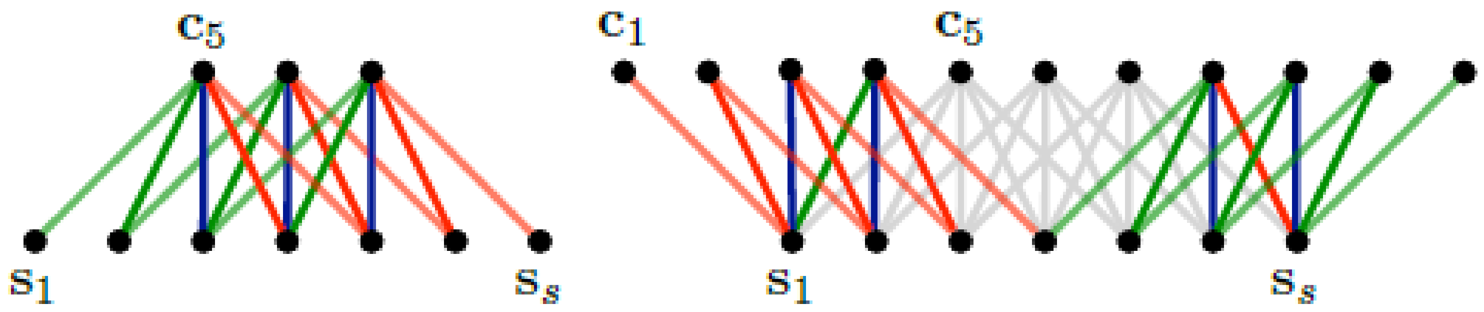
$[\ldots 0, w_1, w_0, 0\ldots]$.

$$y_0 = \sigma(W_0 x_0 + W_1 x_1 - b)$$

$$y_1 = \sigma(W_0 x_1 + W_1 x_2 - b)$$

CNN

- Convolution in 1D
  - Narrow and wide types of convolution.
  - The filter m has size m = 5.

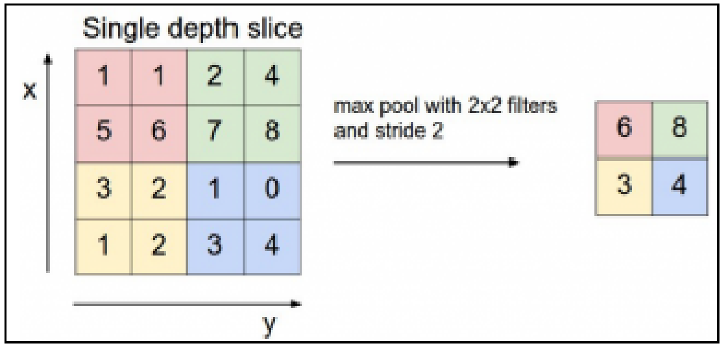


CNN

- Pooling
  - pooling layers
  - Typically applied after the convolutional layers.
  - A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby inputs
  - Pooling layers subsample their input
  - Example: max pooling
  - Several feature maps and sub-sampling
    - Gradual reduction of spatial resolution compensated by increasing no. of features

CNN

- Pooling
  - pooling functions:
    - Max



1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

6	8
3	4

6,8,3,4 are the maximum values in each of the 2X2 regions of same color

    - Average of a rectangular neighborhood
    - L2 norm of a rectangular neighborhood
    - Weighted average based on the distance from the central pixel
    - k-max pooling

CNN

- Basic Convolution & Pooling
  - sequence of words x = $x_1, \ldots, x_n$
  - A 1d convolution layer of width k works by moving a sliding window of size k over the sentence
  - applying the same filter to each window in the sequence $(v(x_i), v(x_{i+1}), \ldots. v(x_{i+k-1}))$
  - The result of the convolution layer is m vectors
  - $P_1, \ldots. p_m$, $p_i \in \mathbb{R}^{d_{conv}}$ where:

$$\mathbf{p_i} = g(\mathbf{w_i}\mathbf{W} + \mathbf{b})$$

CNN

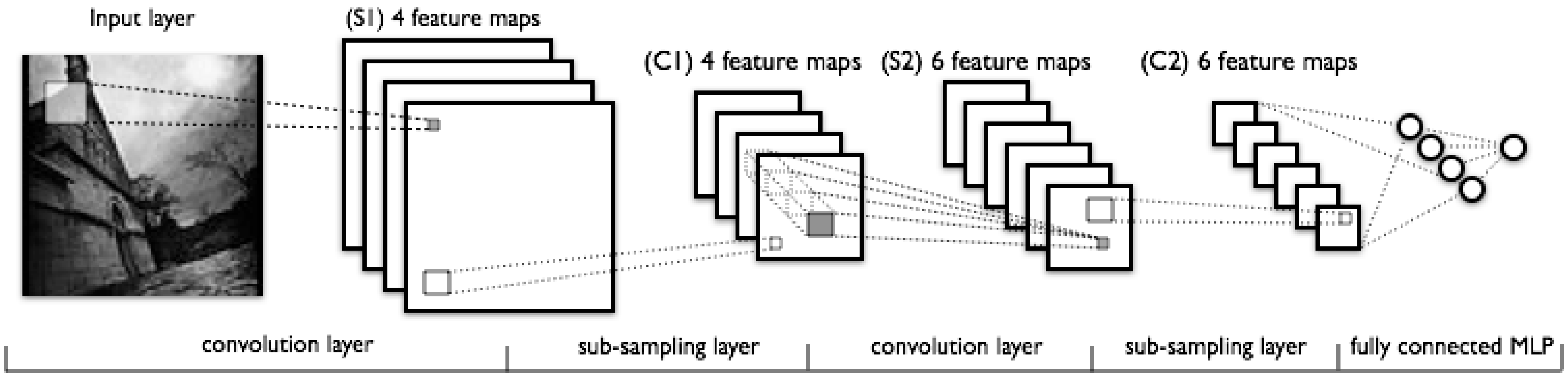
- Basic Convolution & Pooling
  - The m vectors are then combined using a max pooling layer, resulting in a single d_{conv} dimensional vector c.

$$c_j = \max_{1<i\leq m} \mathbf{p_i}[j]$$

  - we can split the vectors p_i into distinct groups, apply the pooling separately on each group, and then concatenate the resulting d_{conv} vectors.

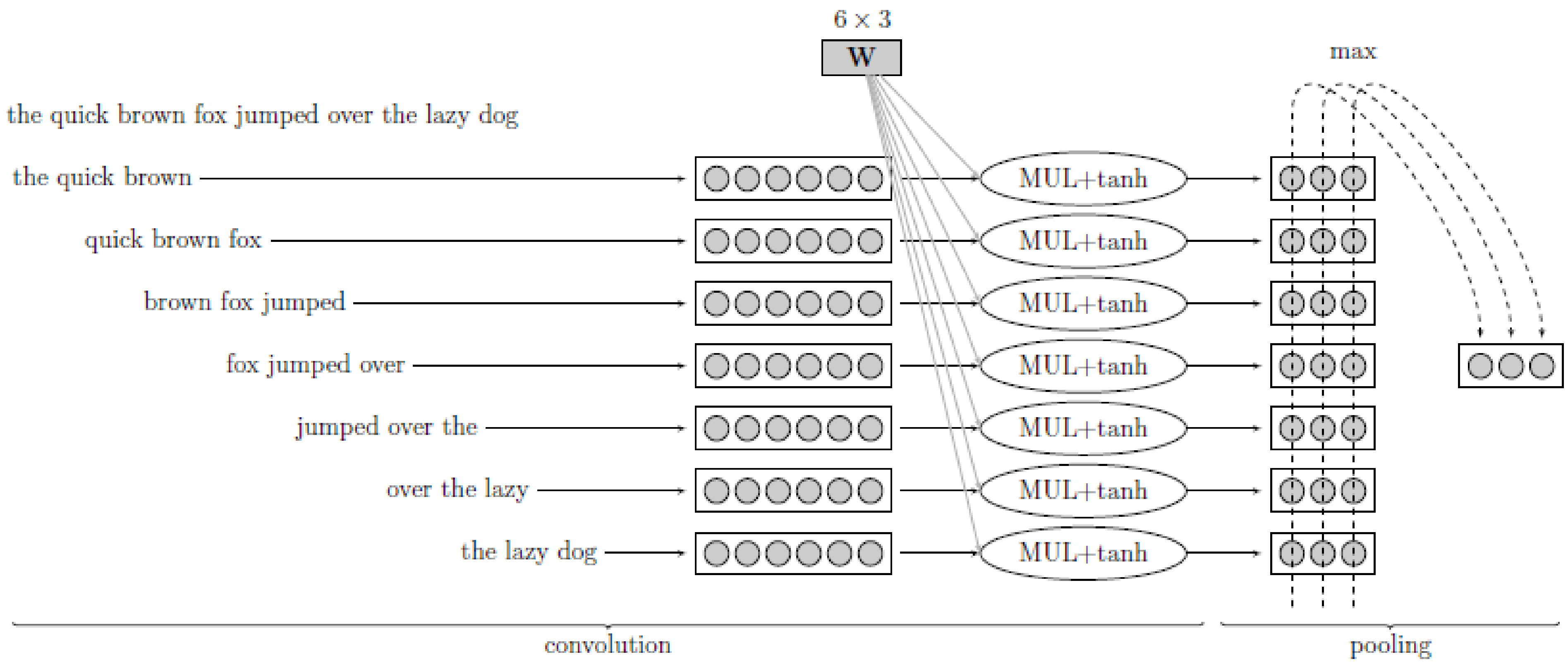
CNN

- Pooling
  - 2 layer convolution & pooling:



CNN

- Basic Convolution & Pooling



CNN with Keras

```
from keras.models import Input, Model, Sequential
from keras.optimizers import RMSprop
```

In Keras, you would use a 1D convnet via the Conv1D layer.
There exist also, Conv2D and Conv3D.
It takes as input 3D tensors with shape (samples, time, features)
It returns similarly-shaped 3D tensors.
The convolution window is a 1D window on the temporal axis, axis 1 in the input tensor.

CNN with Keras

```
def buildKerasConv1Model(parameters, x_train, y_train, x_test, y_test, epochs):
    global history
    input_shape = (x_train.shape[1],x_train.shape[2])
    model = Sequential()
    model.add(layers.Conv1D(32, 7, activation='relu', input_shape = input_shape))
    model.add(layers.MaxPooling1D(5))
    model.add(layers.Conv1D(32, 7, activation='relu'))
    model.add(layers.GlobalMaxPooling1D())
    model.add(layers.Dense(1))
    model.summary()
    model.compile(optimizer=RMSprop(lr=1e-4),loss='binary_crossentropy', metrics=['acc'])
    history = model.fit(x_train, y_train, epochs=epochs, batch_size=128, validation_split=0.2)
    return model
```

History allows a further analysis of the process for adjusting metaparameters

CNN with Keras

```
def buildKerasConv1Model(parameters, x_train, y_train, x_test, y_test, epochs):
    global history
    input_shape = (x_train.shape[1],x_train.shape[2])
    model = Sequential()
    model.add(layers.Conv1D(32, 7, activation='relu', input_shape = input_shape))
    model.add(layers.MaxPooling1D(5))
    model.add(layers.Conv1D(32, 7, activation='relu'))
    model.add(layers.GlobalMaxPooling1D())
    model.add(layers.Dense(1))
    model.summary()
    model.compile(optimizer=RMSprop(lr=1e-4),loss='binary_crossentropy', metrics=['acc'])
    history = model.fit(x_train, y_train, epochs=epochs, batch_size=128, validation_split=0.2)
    return model
```

Input_shape consists of the number of vectors and their dimensión
The input shape is one of the parameters of the first conv1 layer

Sequential model is just a sequence of layers
that have to be added to the model

CNN with Keras

```
def buildKerasConv1Model(parameters, x_train, y_train, x_test, y_test, epochs):
    global history
    input_shape = (x_train.shape[1],x_train.shape[2])
    model = Sequential()
    model.add(layers.Conv1D(32, 7, activation='relu', input_shape = input_shape))
    model.add(layers.MaxPooling1D(5))
    model.add(layers.Conv1D(32, 7, activation='relu'))
    model.add(layers.GlobalMaxPooling1D())
    model.add(layers.Dense(1))
    model.summary()
    model.compile(optimizer=RMSprop(lr=1e-4),loss='binary_crossentropy', metrics=['acc'])
    history = model.fit(x_train, y_train, epochs=epochs, batch_size=128, validation_split=0.2)
    return model
```

The model consists of two convolutional layers followed each by a MaxPooling layer.

eventually ending in either a global pooling layer or a Flatten layer, turning the 3D outputs into 2D outputs, allowing to add one or more Dense layers to the model, for classification or regression.

CNN with Keras

```
def buildKerasConv1Model(parameters, x_train, y_train, x_test, y_test, epochs):
    global history
    input_shape = (x_train.shape[1],x_train.shape[2])
    model = Sequential()
    model.add(layers.Conv1D(32, 7, activation='relu', input_shape = input_shape))
    model.add(layers.MaxPooling1D(5))
    model.add(layers.Conv1D(32, 7, activation='relu'))
    model.add(layers.GlobalMaxPooling1D())
    model.add(layers.Dense(1))
    model.summary()
    model.compile(optimizer=RMSprop(lr=1e-4),loss='binary_crossentropy', metrics=['acc'])
    history = model.fit(x_train, y_train, epochs=epochs, batch_size=128, validation_split=0.2)
    return model
```

The model is summarized, compiled and learned (fit)

CNN with Keras

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_1 (Conv1D)            (None, 74, 32)            448704
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 14, 32)            0
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 8, 32)             7200
_________________________________________________________________
global_max_pooling1d_1       (Glob (None, 32)          0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 33
=================================================================
Total params: 455,937
Trainable params: 455,937
Non-trainable params: 0
```

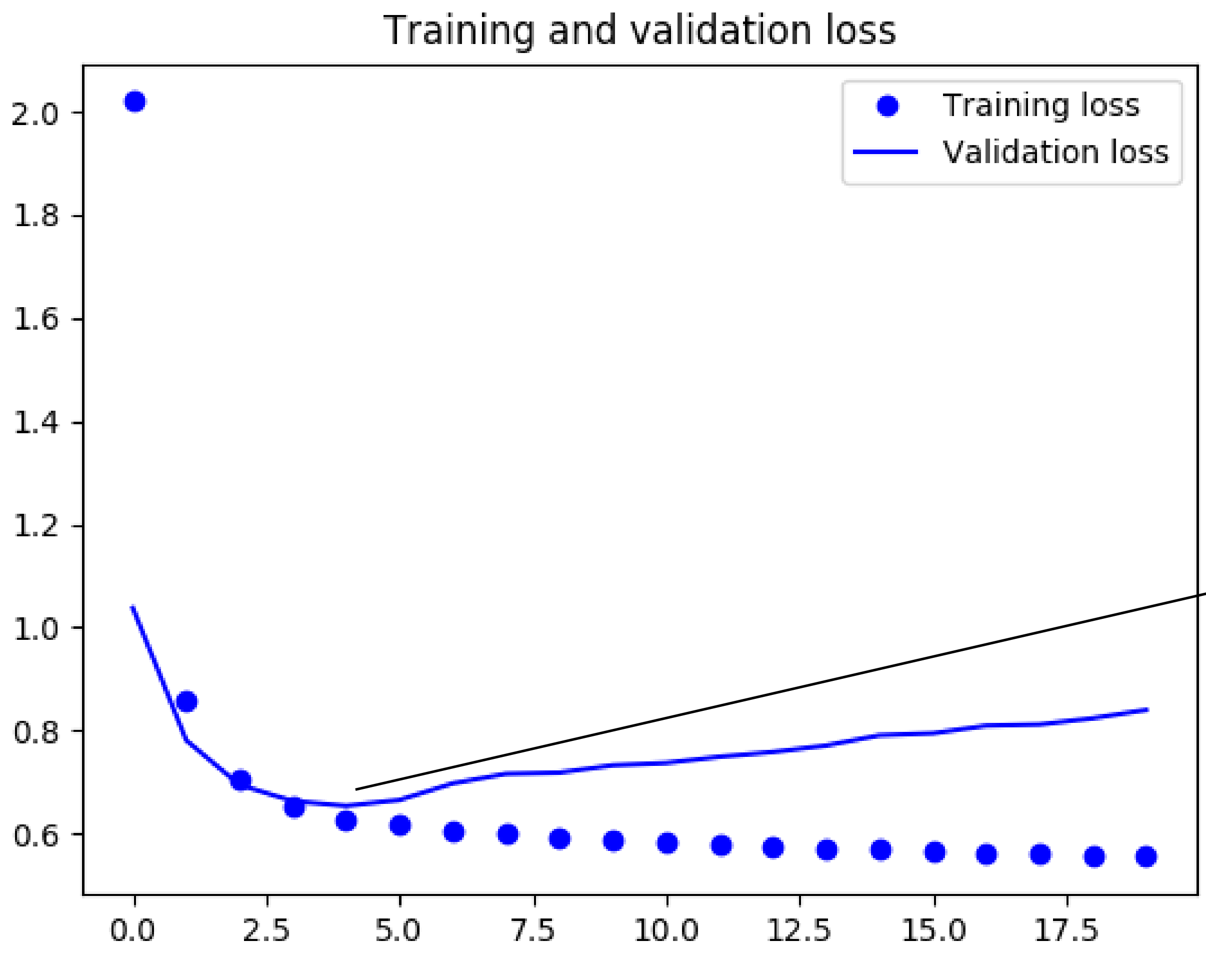
CNN with Keras

```
Train on 5760 samples, validate on 1440 samples
Epoch 1/10 142s - loss: 0.9316 - acc: 0.6250 - val_loss: 0.7563 - val_acc: 0.6368
Epoch 2/10 91s - loss: 0.6910 - acc: 0.6250 - val_loss: 0.6820 - val_acc: 0.6174
Epoch 3/10 98s - loss: 0.6467 - acc: 0.6337 - val_loss: 0.6636 - val_acc: 0.6139
Epoch 4/10 89s - loss: 0.6291 - acc: 0.6432 - val_loss: 0.6575 - val_acc: 0.6229
Epoch 5/10 92s - loss: 0.6168 - acc: 0.6556 - val_loss: 0.6781 - val_acc: 0.6382
Epoch 6/10 89s - loss: 0.6077 - acc: 0.6623 - val_loss: 0.6800 - val_acc: 0.6278
Epoch 7/10 94s - loss: 0.5997 - acc: 0.6635 - val_loss: 0.6792 - val_acc: 0.6417
Epoch 8/10 93s - loss: 0.5926 - acc: 0.6679 - val_loss: 0.6984 - val_acc: 0.6306
Epoch 9/10 94s - loss: 0.5872 - acc: 0.6734 - val_loss: 0.7079 - val_acc: 0.6292
Epoch 10/10 97s - loss: 0.5811 - acc: 0.6750 - val_loss: 0.7100 - val_acc: 0.6326
```

Extending to 20 epochs

```
92s - loss: 0.5551 - acc: 0.6898 - val_loss: 0.8395 - val_acc: 0.6312
```

CNN with Keras



Clear overfitting

NN models for NLP

- Feed-forward neural Networks (FFNM)
- Convolutional NN (CNN)
- Recurrent NN (RNN)
- Autoencoders
- Recursive NN (RecNN)
- Gated models
  - Long Short Time Models (LSTM)
  - Gated Recurrent Units (GRU)
- Attention-based models

RNN models

- Recurrent NN (RNN)
  - representing arbitrarily sized structured inputs (e.g.sentences) in a fixed-size vector, while paying attention to the structured properties of the input.
  - The hidden layer activations are computed by iterating the following equations from t = 1 to T and from n = 2 to N:

$$h_t^1 = \mathcal{H}\left(W_{ih^1} x_t + W_{h^1 h^1} h_{t-1}^1 + b_h^1\right)$$

$$h_t^n = \mathcal{H}\left(W_{ih^n} x_t + W_{h^{n-1} h^n} h_t^{n-1} + W_{h^n h^n} h_{t-1}^n + b_h^n\right)$$

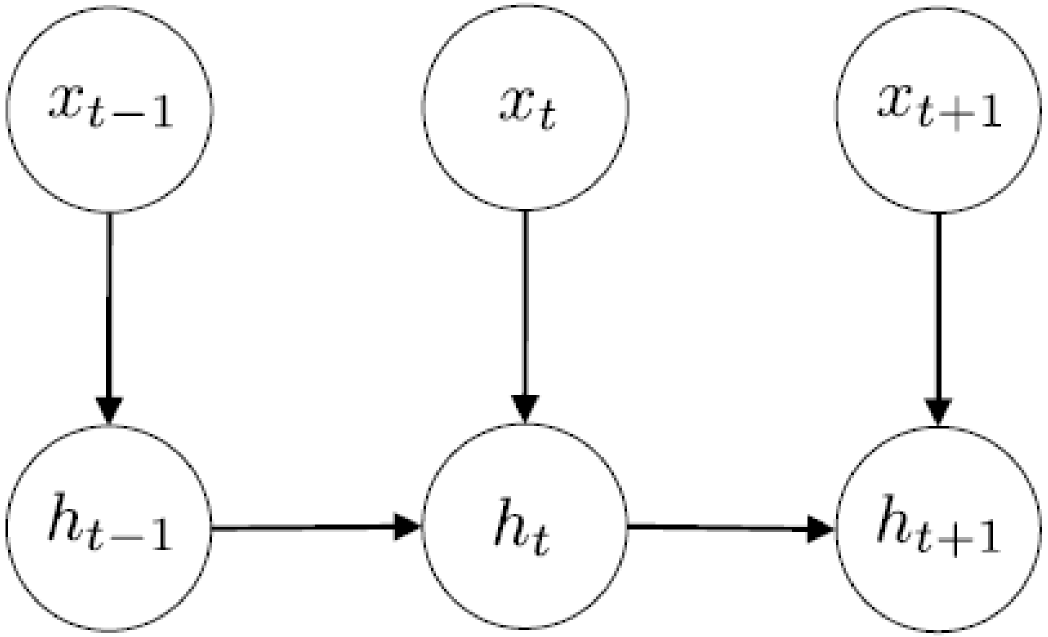
$$\hat{y}_t = b_y + \sum_{n=1}^{N} W_{h^n y} h_t^n$$

$$y_t = \mathcal{Y}(\hat{y}_t)$$

RNN models

$$h_t = \tanh\left(W^{(xh)}x_t + W^{(hh)}h_{t-1} + b^{(h)}\right)$$

“hidden vector”

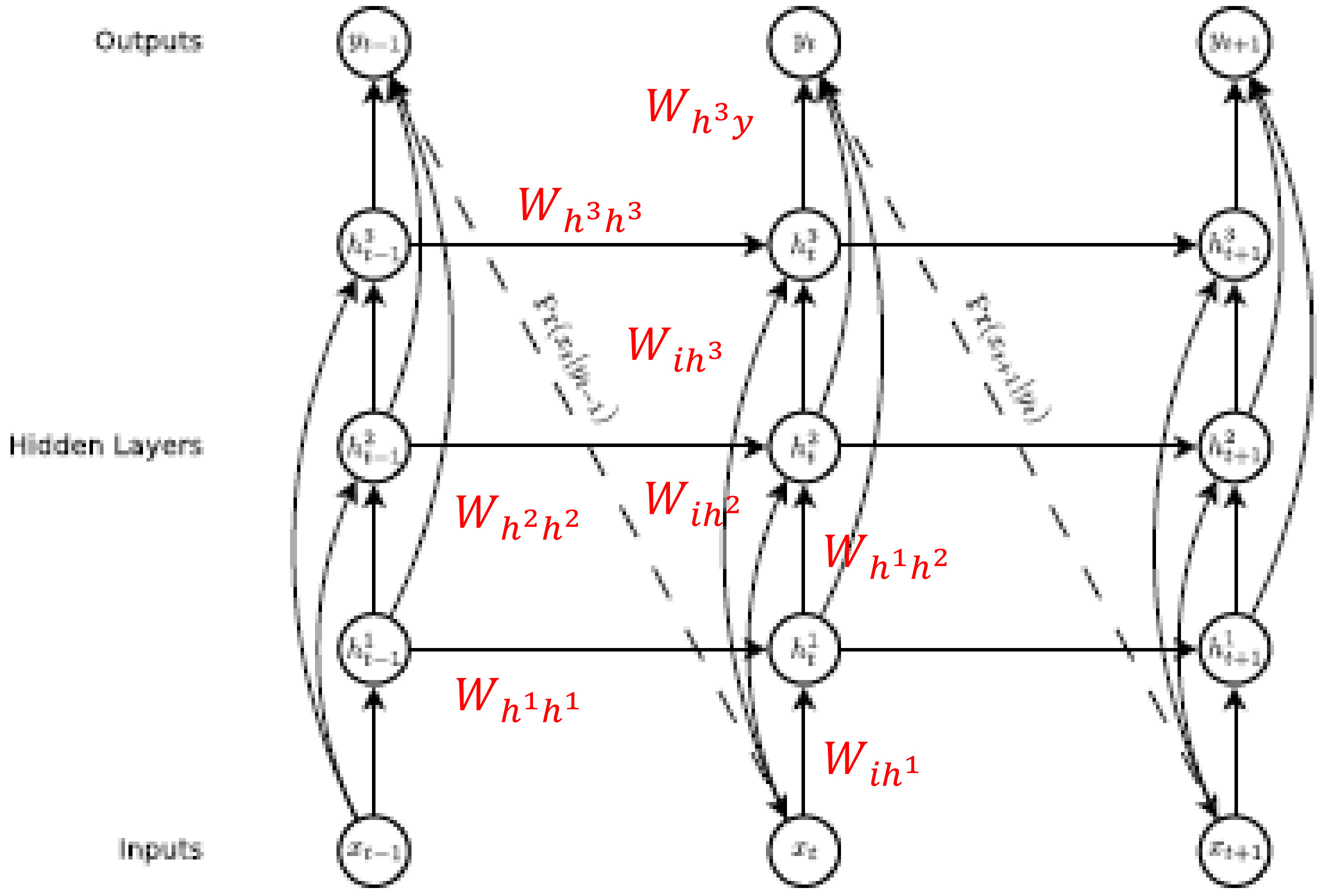


$$\begin{tikzcd} x_{t-1} \arrow[d] & x_t \arrow[d] & x_{t+1} \arrow[d] \\ h_{t-1} \arrow[r] & h_t \arrow[r] & h_{t+1} \end{tikzcd}$$

RNN models

- Stacked RNN
  - It is sometimes useful to stack several recurrent layers one after the other in order to increase the representational power of a network. In such a setup, you have to get all intermediate layers to return full sequences.

RNN



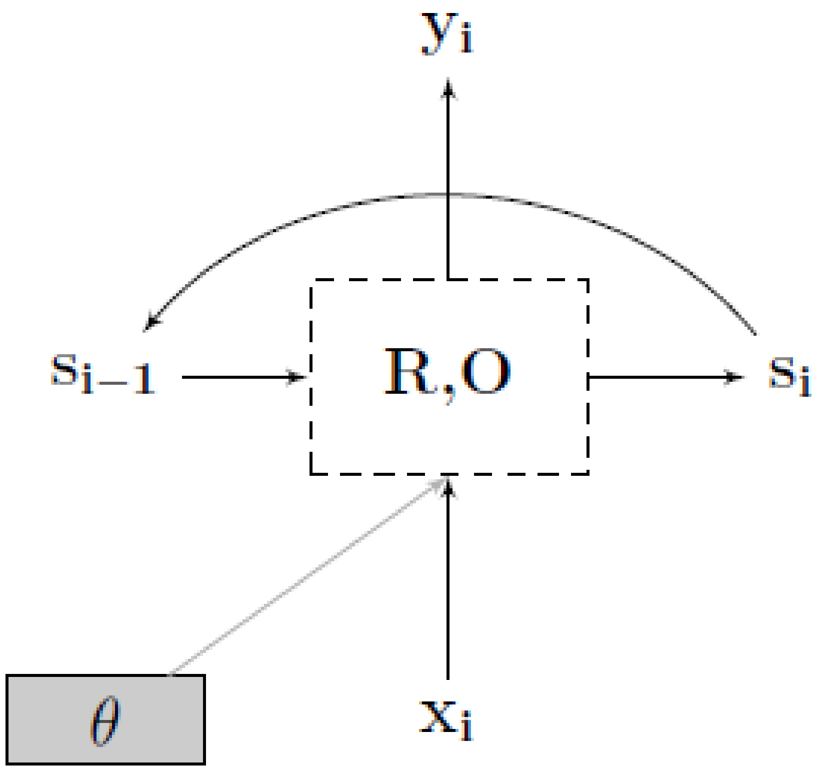
Outputs
Hidden Layers
Inputs
y_{t-1}
y_t
y_{t+1}
h^3_{t-1}
h^3_t
h^3_{t+1}
h^2_{t-1}
h^2_t
h^2_{t+1}
h^1_{t-1}
h^1_t
h^1_{t+1}
x_{t-1}
x_t
x_{t+1}
$\Pr(x_t|y_{t-1})$
$\Pr(x_{t+1}|y_t)$
W_{h^3y}
$W_{h^3h^3}$
W_{ih^3}
$W_{h^2h^2}$
W_{ih^2}
$W_{h^1h^2}$
$W_{h^1h^1}$
W_{ih^1}

RNN models

- Recurrent NN (RNN)
  - An input vector sequence $x = (x_1, \ldots, x_T)$ is passed through weighted connections to a stack of N recurrently connected hidden layers to compute first the hidden vector sequences $h^n = (h^n_1, \ldots, h^n_T)$ and then the output vector sequence $y = (y_1, \ldots, y_T)$. Each output vector y_t is used to parameterise a predictive distribution $Pr(x_{t+1}|y_t)$ over the possible next inputs x_{t+1}. The first element x_1 of every input sequence is always a null vector whose entries are all zero; the network therefore emits a prediction for x_2, the first real input, with no prior information. The network is `deep' in both space and time, in the sense that every piece of information passing either vertically or horizontally through the computation graph will be acted on by multiple successive weight matrices and nonlinearities.

RNN models

- Recurrent NN (RNN)

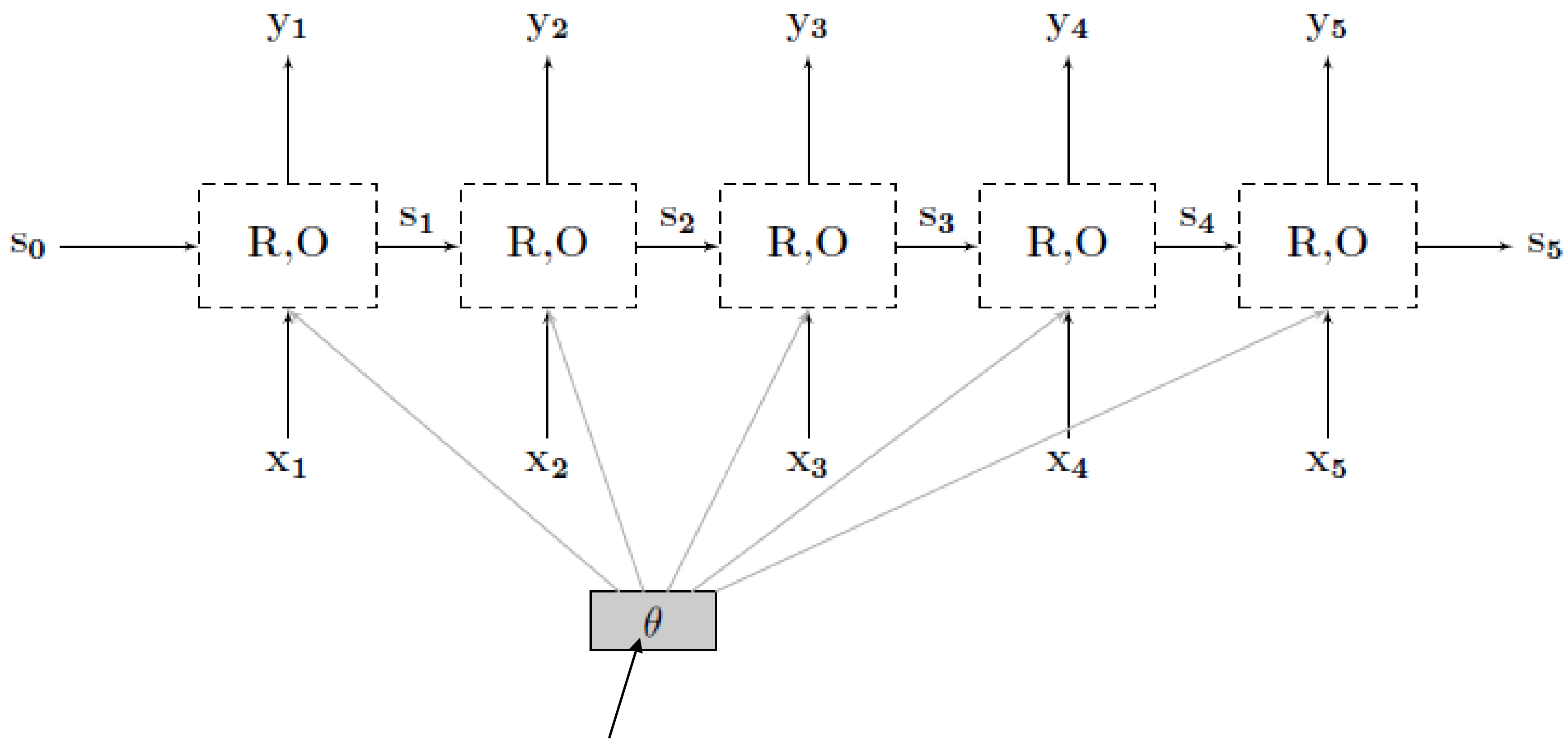


RNN models

- Simple RNN architecture
  - The state at position i is a linear combination of the input at position i and the previous state, passed through a non-linear activation (commonly tanh or ReLU). The output at position i is the same as the hidden state in that position.

RNN models

- Recurrent NN (RNN)

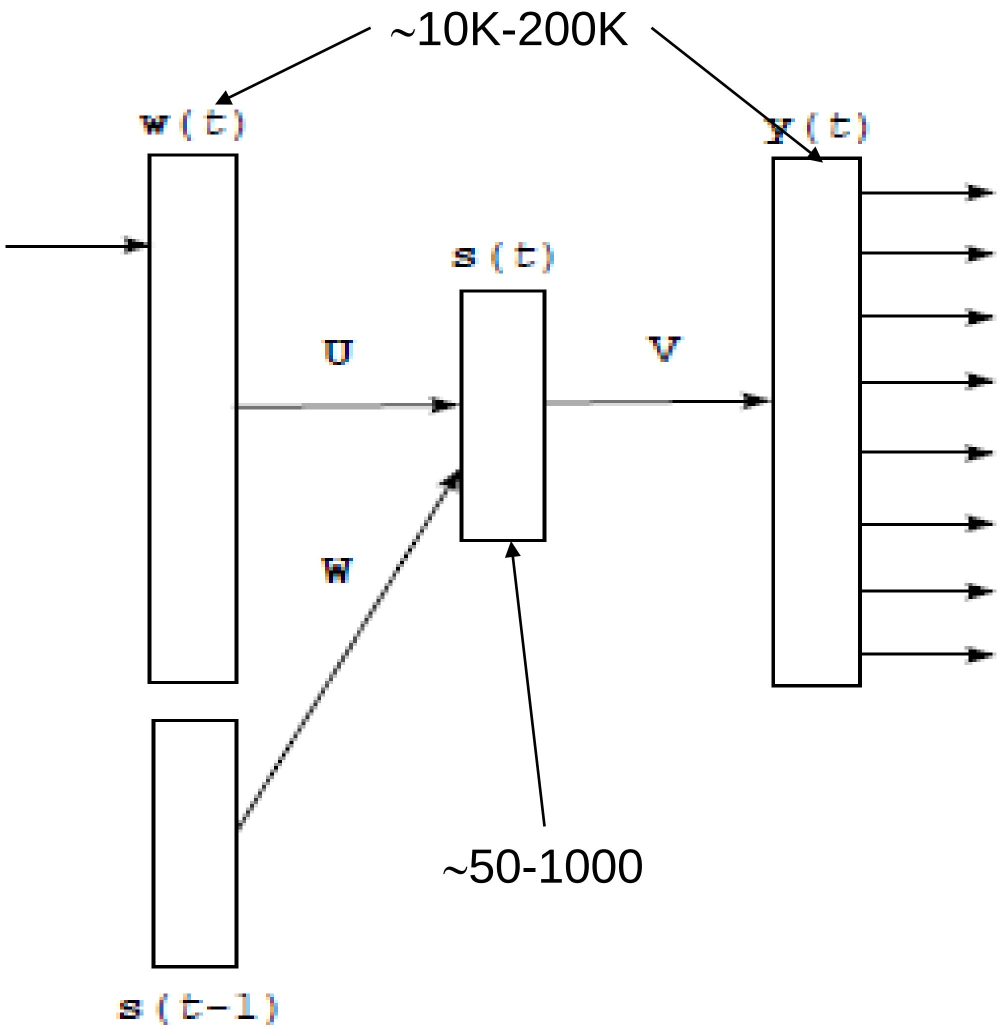


Same parameterization

RNN models

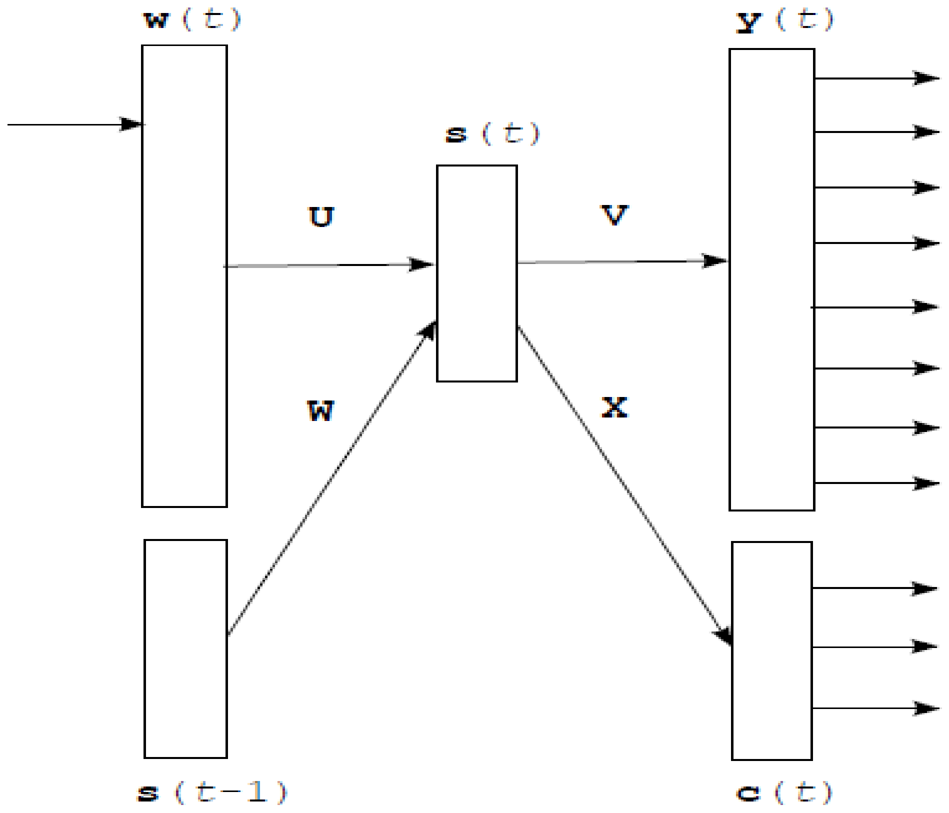
RNN language models

Without W bigram NN LM
s(t) = f(Uw(t) + Ws(t-1))
f sigmoid or tanh
y(t) = g(Vs(t))
g SoftMax



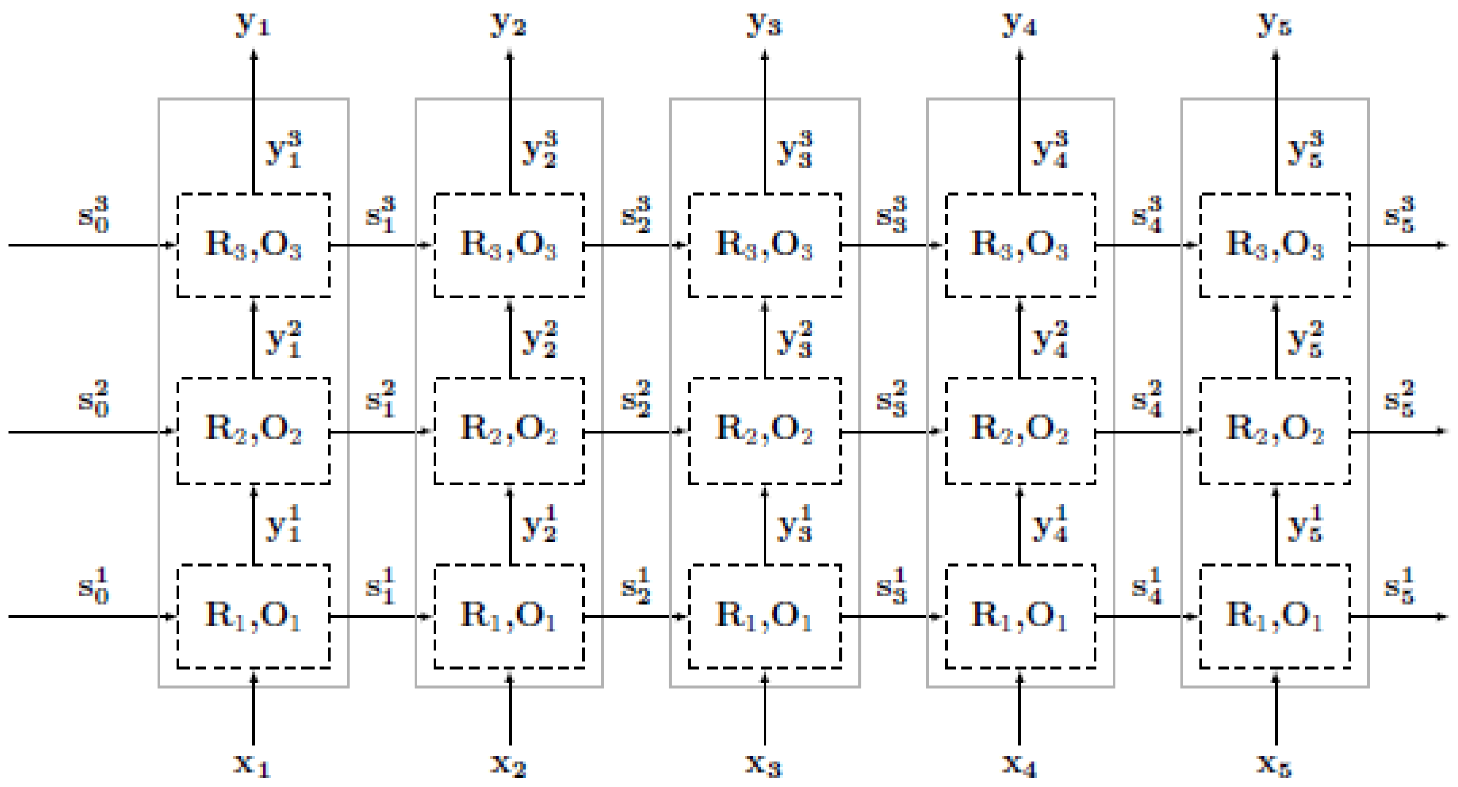
RNN models

Factorization of the output layer,
c(t) is the class layer.
Assignment word to class



RNN models

- Multi-layer (stacked) RNNs

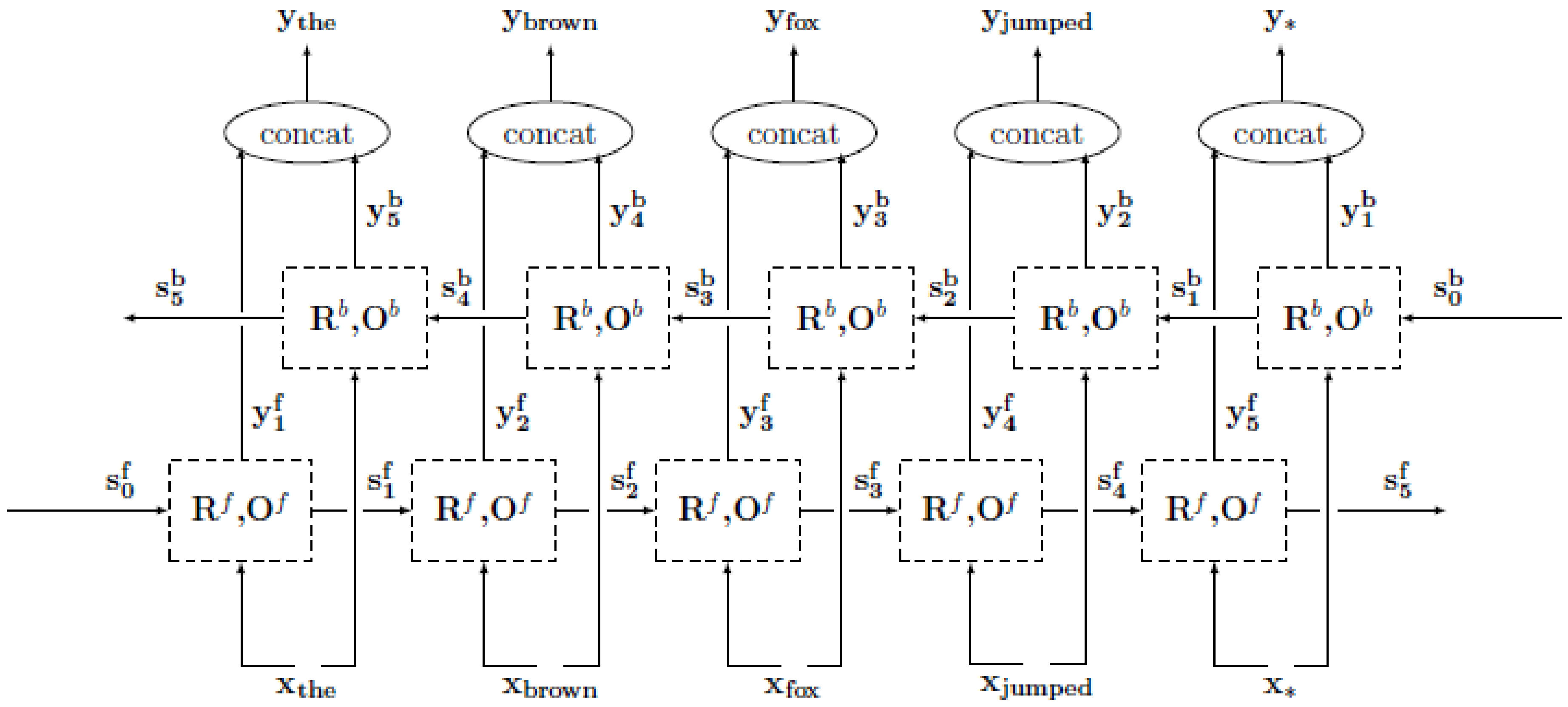


RNN models

- bidirectional-RNN (BI-RNN)
  - Much like the RNN relaxes the Markov assumption and allows looking arbitrarily back into the past, the BI-RNN relaxes the fixed window size assumption, allowing to look arbitrarily far at both the past and the future.
  - Two separate states, s_i^f and s_i^b for each input position i. The forward state s_i^f is based on x_1, x_2, …, x_i, while the backward state s_i^b is based on x_n, x_{n-1}, …, x_i.

RNN models

- bidirectional-RNN (BI-RNN)



RNN with Keras

```
from keras.layers import SimpleRNN
from keras.layers import Activation
from keras import initializers
```

SimpleRNN processes batches of sequences, like all other Keras layers, not just a single sequence. This means that it takes inputs of shape (batch_size, timesteps, input_features), rather than (timesteps, input_features).
SimpleRNN can be run in two different modes: it can return either the full sequences of successive outputs for each timestep (a 3D tensor of shape (batch_size, timesteps, output_features)), or it can return only the last output for each input sequence (a 2D tensor of shape (batch_size, output_features)). These two modes are controlled by the return_sequences constructor argument.

RNN with Keras

```
def buildKerasSimpleRNNModel(parameters, x_train, y_train, x_test, y_test, epochs):
    global history
    input_shape = (x_train.shape[1],x_train.shape[2]); print 'input_shape', input_shape
    hidden_units = 100; learning_rate = 1e-6
    model = Sequential()
    model.add(SimpleRNN( hidden_units, input_shape=input_shape,
          kernel_initializer=initializers.RandomNormal(stddev=0.001),
          recurrent_initializer=initializers.Identity(gain=1.0), activation='relu',))
    model.add(Dense(1))
    model.add(Activation('softmax')); rmsprop = RMSprop(lr=learning_rate)
    model.summary()
    model.compile(loss='binary_crossentropy', optimizer=rmsprop, metrics=['accuracy'])
    history = model.fit(x_train, y_train, batch_size=32,
        epochs=epochs, verbose=2, validation_data=(x_test, y_test))
    return model
```

Dimensionality of the output space

One line per epoch

Number of samples per gradient update

RNN with Keras

```
def buildKerasSimpleRNNModel(parameters, x_train, y_train, x_test, y_test, epochs):
    global history
    input_shape = (x_train.shape[1],x_train.shape[2]); print 'input_shape', input_shape
    hidden_units = 100; learning_rate = 1e-6
    model = Sequential()
    model.add(SimpleRNN( hidden_units, input_shape=input_shape,
        kernel_initializer=initializers.RandomNormal(stddev=0.001),
        recurrent_initializer=initializers.Identity(gain=1.0), activation='relu',))
    model.add(Dense(1))
    model.add(Activation('softmax')); rmsprop = RMSprop(lr=learning_rate)
    model.summary()
    model.compile(loss='binary_crossentropy', optimizer=rmsprop, metrics=['accuracy'])
    history = model.fit(x_train, y_train, batch_size=32,
        epochs=epochs, verbose=2, validation_data=(x_test, y_test))
    return model
```

Initializer for the kernel weight matrix

Optimizers: Adam, RMSprop, Nadam, Adadelta, SGD, Adagrad, Adamax

Initializer for the recurrent_kernel weight matrix

List of metrics to be evaluated during training and testing

NN models for NLP

- Feed-forward neural Networks (FFNM)
- Convolutional NN (CNN)
- Recurrent NN (RNN)
- Autoencoders
- Recursive NN (RecNN)
- Gated models
  - Long Short Time Models (LSTM)
  - Gated Recurrent Units (GRU)
- Attention-based models

Autoencoders

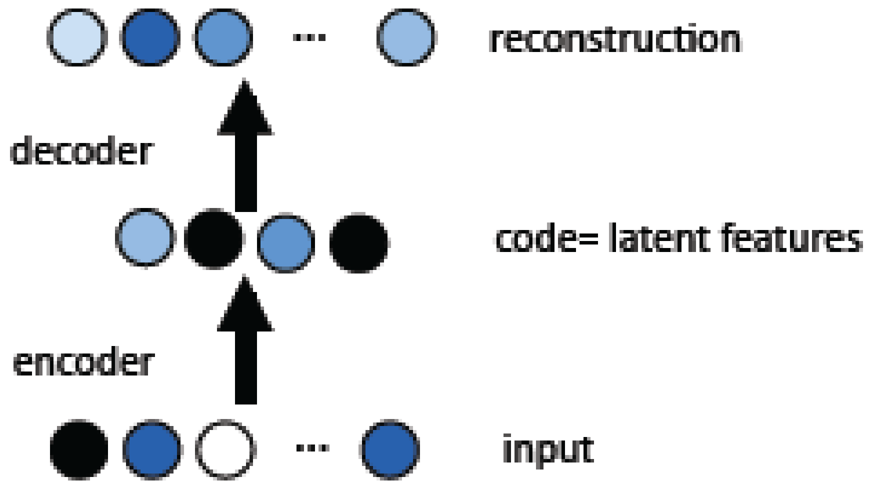
- An autoencoder takes an input $\mathbf{x} \in [0,1]^d$ and first maps it (with an **encoder**) to a hidden representation $\mathbf{y} \in [0,1]^{d'}$ through a deterministic mapping, e.g.: $\mathbf{y} = \sigma(\mathbf{Wx} + \mathbf{b})$
- Where σ is a non-linearity such as the sigmoid. The latent representation **y**, or code, is then mapped back (with a decoder) into a reconstruction **z** of the same shape as **x**. The mapping happens through a similar transformation, e.g.: $\mathbf{z} = \sigma(\mathbf{W'y} + \mathbf{b'})$
- Optionally, the weight matrix **W'** of the reverse mapping may be constrained to be the transpose of the forward mapping: $\mathbf{W'} = \mathbf{W}^T$. This is referred to as tied weights.
- The parameters **W**, **b**, **W'**, **b'** are optimized such that the average reconstruction error is minimized.
- The reconstruction error can be measured in many ways, depending on the appropriate distributional assumptions on the input given the code (squared error, cross-entropy, …).

Autoencoders

$y = \tanh(Wx+b)$

$x' = \tanh(W^T y + b')$

Cost of reconstruction
$= ||x' - x||^2$



Autoencoders

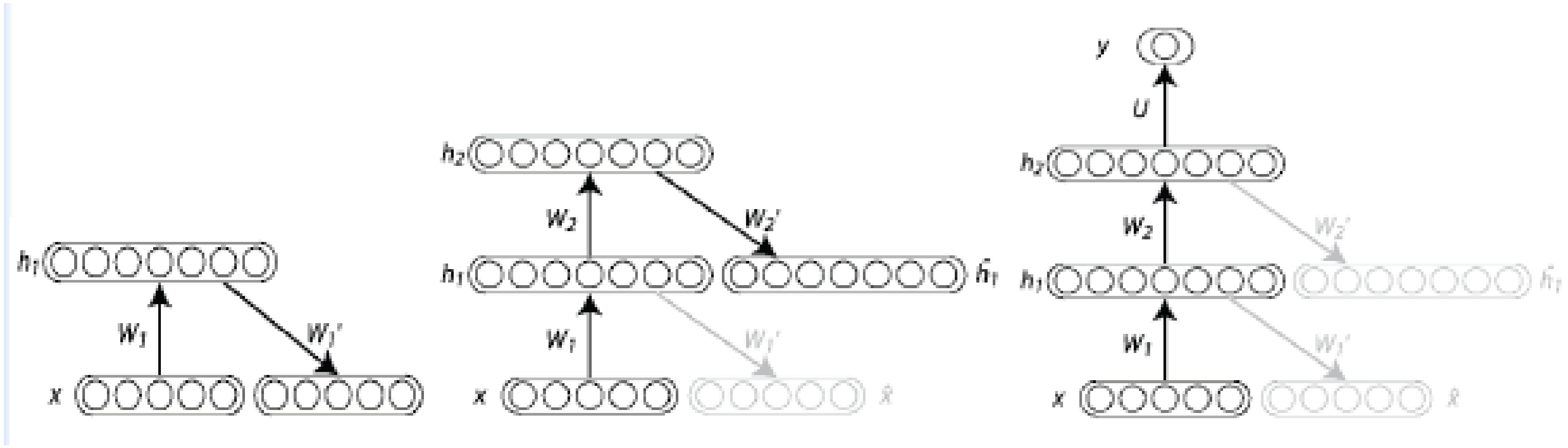
- Denoising Autoencoders (dAE)
  - The dAE is a stochastic version of the auto-encoder.
  - The idea behind dAE is simple. In order to force the hidden layer to discover more robust features and prevent it from simply learning the identity, we train the autoencoder to ***reconstruct the input from a corrupted version of it*****.**
  - Intuitively, a dAE does two things: try to encode the input (preserve the information about the input), and try to undo the effect of a corruption process stochastically applied to the input of the auto-encoder. The latter can only be done by capturing the statistical dependencies between the inputs.
  - the ***stochastic corruption process*** randomly sets some of the inputs (as many as half of them) to zero. Hence the denoising auto-encoder is trying to *predict the corrupted (i.e. missing) values from the uncorrupted (i.e., non-missing) values*, for randomly selected subsets of missing patterns.

Autoencoders

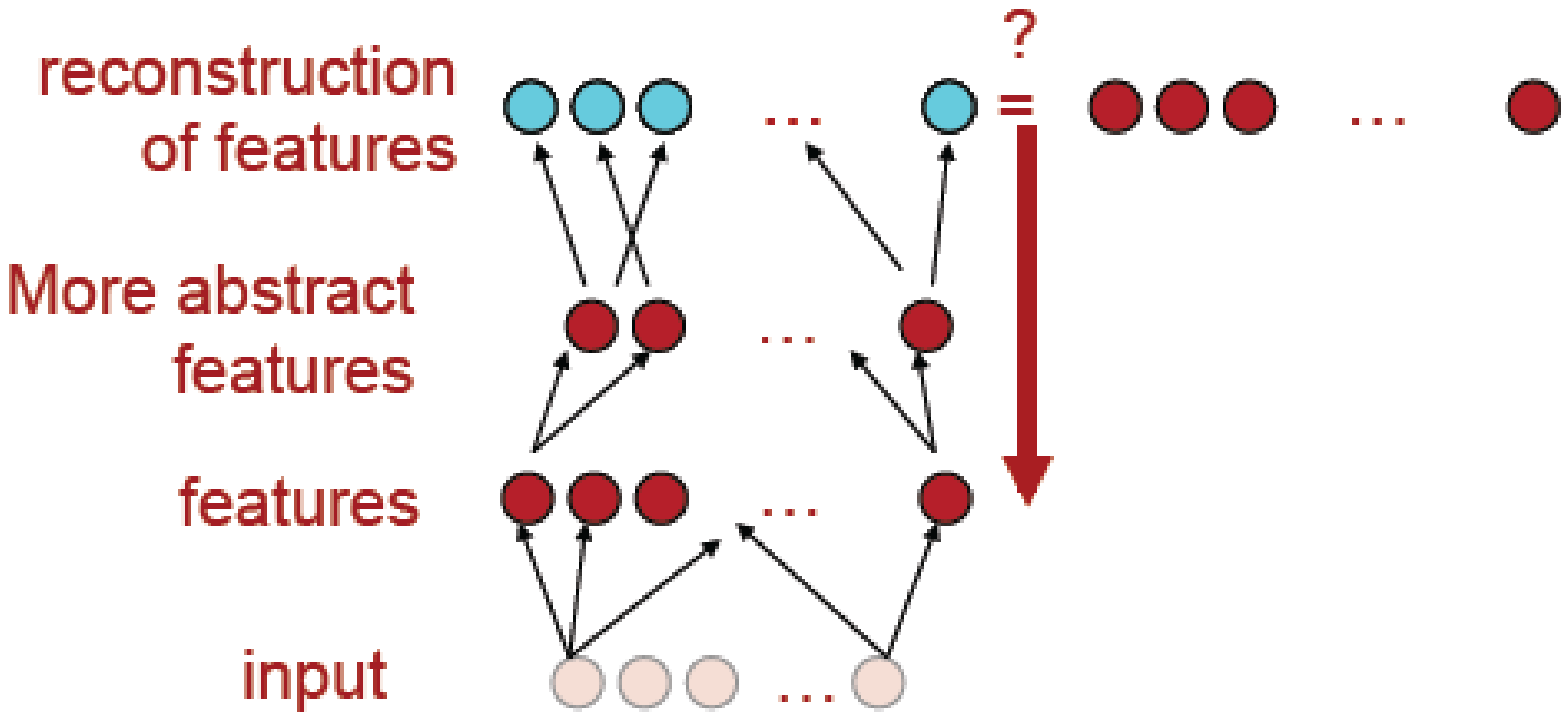
- Recursive Autoencoders (RAE)
  - Combines RecNN with dAE
  - From Socher et al, 2011, used for paraphrase detection (see later)
  - A RAE is a multilayer recursive neural network with input = output
  - It learns feature representations for each node in the tree such that the word vectors underneath each node can be recursively reconstructed.
  - Reconstruction = decoder(encoder(input))

Recursive Autoencoders

- Stacking RAE
  - RAE can be stacked to form highly non-linear representations



Recursive Autoencoders



Autoencoders

- Variational Autoencoders (VAE)
  - augment autoencoders with a little bit of statistics that forces them to learn continuous, highly structured latent spaces.
  - A VAE, instead of compressing **x** into a fixed code in the latent space, turns it into the parameters of a statistical distribution: a mean and a variance. Essentially, this means that we are assuming that **x** has been generated by a statistical process, and that the randomness of this process should be taken into accounting during encoding and decoding. The VAE then uses the mean and variance parameters to randomly sample one element of the distribution, and decodes that element back to the original input. The stochasticity of this process improves robustness and forces the latent space to encode meaningful representations everywhere, i.e. every point sampled in the latent will be decoded to a valid output.

Autoencoders

- Variational Autoencoders (VAE)
  - First, an encoder module turns the input samples **x** into two parameters in a latent space of representations, which we will note z_mean and z_log_variance. Then, we randomly sample a point **z** from the latent normal distribution that is assumed to generate the input image, via z = z_mean + exp(z_log_variance) * epsilon. where epsilon is a random tensor of small values. Finally, a decoder module will map this point in the latent space back to the original input.
  - The parameters of a VAE are trained via two loss functions: first, a reconstruction loss that forces the decoded samples to match the initial inputs, and a regularization loss, which helps in learning well-formed latent spaces and reducing overfitting to the training data.

NN models for NLP

- Feed-forward neural Networks (FFNM)
- Convolutional NN (CNN)
- Recurrent NN (RNN)
- Autoencoders
- Recursive NN (RecNN)
- Gated models
  - Long Short Time Models (LSTM)
  - Gated Recurrent Units (GRU)
- Attention-based models

RecNN

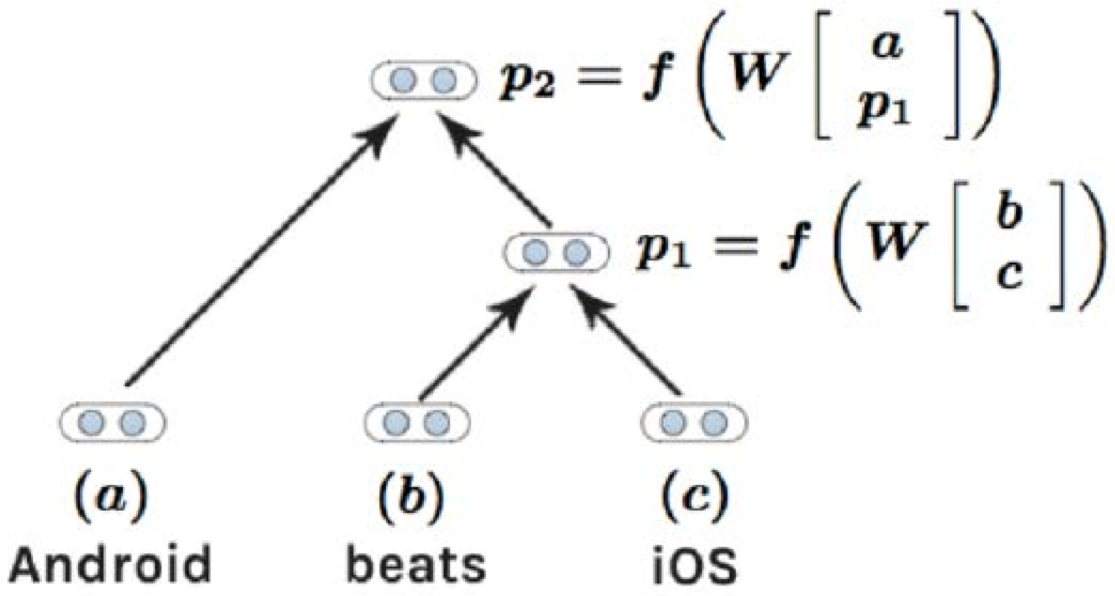
- Recursive Neural Networks
  - See Goldberg 2016
  - The RNN is very useful for modeling sequences. In language processing, it is often natural and desirable to work with tree structures.
  - The trees can be syntactic trees, discourse trees, or even trees representing the sentiment expressed by various parts of a sentence.
  - RecNN are a generalization of the RNN from sequences to (binary) trees.
  - Much like the RNN encodes each sentence prefix as a state vector, the RecNN encodes each tree-node as a state vector in R^d.

RecNN

$$\begin{aligned}
\text{RecNN}(x_1, \ldots, x_n, \mathcal{T}) =& \{\mathbf{s}^{\mathbf{A}}_{\mathbf{i:j}} \in \mathbb{R}^d \mid q^A_{i:j} \in \mathcal{T}\} \\
\mathbf{s}^{\mathbf{A}}_{\mathbf{i:i}} =& v(x_i) \\
\mathbf{s}^{\mathbf{A}}_{\mathbf{i:j}} =& R(A, B, C, \mathbf{s}^{\mathbf{B}}_{\mathbf{i:k}}, \mathbf{s}^{\mathbf{C}}_{\mathbf{k+1:j}}) \qquad q^B_{i:k} \in \mathcal{T}, \;\; q^C_{k+1:j} \in \mathcal{T}
\end{aligned}$$

RecNN

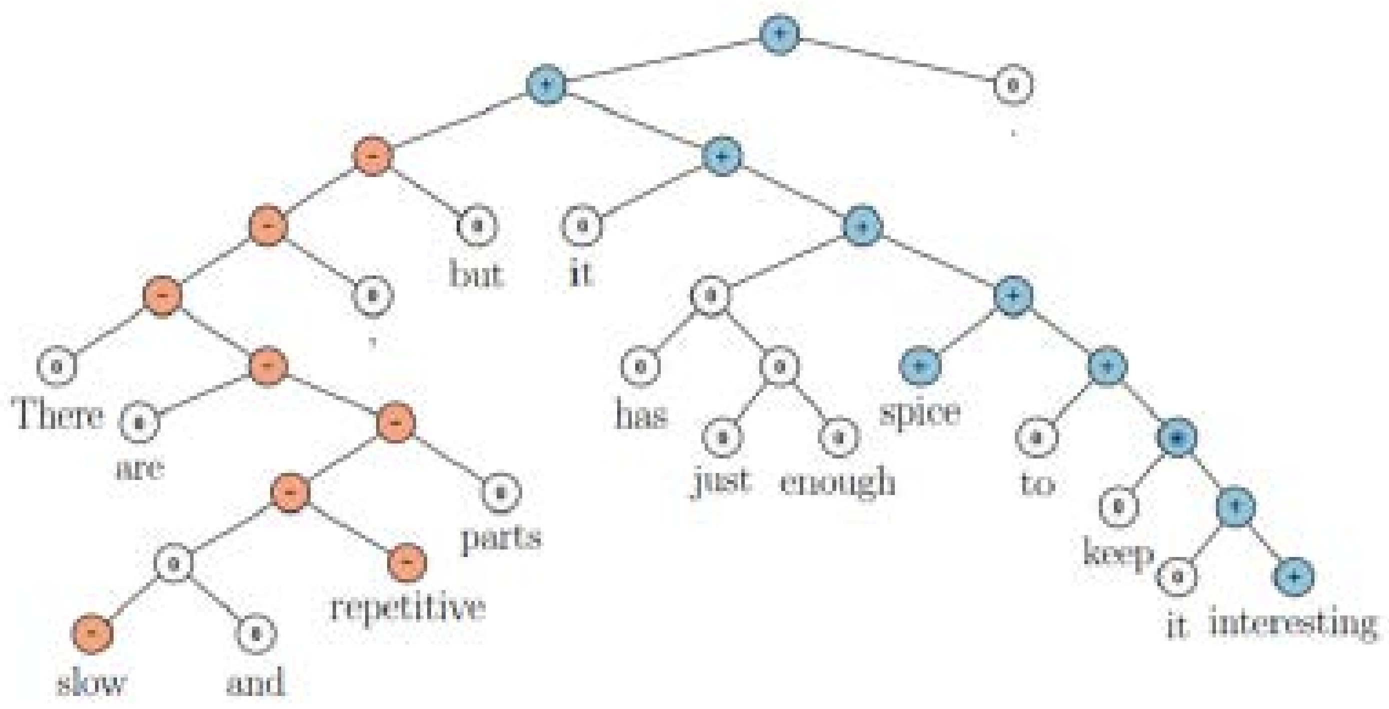
- Recursive Neural Networks for NLP
  - first, run a constituent parser on the sentence
  - convert the constituent tree to a binary tree
  - construct vector for sentence recursively at each rewrite ("split point"):



$p_2 = f\left(W\begin{bmatrix} a \\ p_1 \end{bmatrix}\right)$
$p_1 = f\left(W\begin{bmatrix} b \\ c \end{bmatrix}\right)$
(a)
Android
(b)
beats
(c)
iOS

RecNN

- Recursive Neural Networks for NLP
  - Recursive neural networks applied on a sentence for sentiment classification. Note that “but” plays a crucial role on determining the sentiment of the whole sentence (Socher et al. 2013)



NN models for NLP

- Feed-forward neural Networks (FFNM)
- Convolutional NN (CNN)
- Recurrent NN (RNN)
- Autoencoders
- Recursive NN (RecNN)
- Gated models
  - Long Short Time Models (LSTM)
  - Gated Recurrent Units (GRU)
- Attention-based models

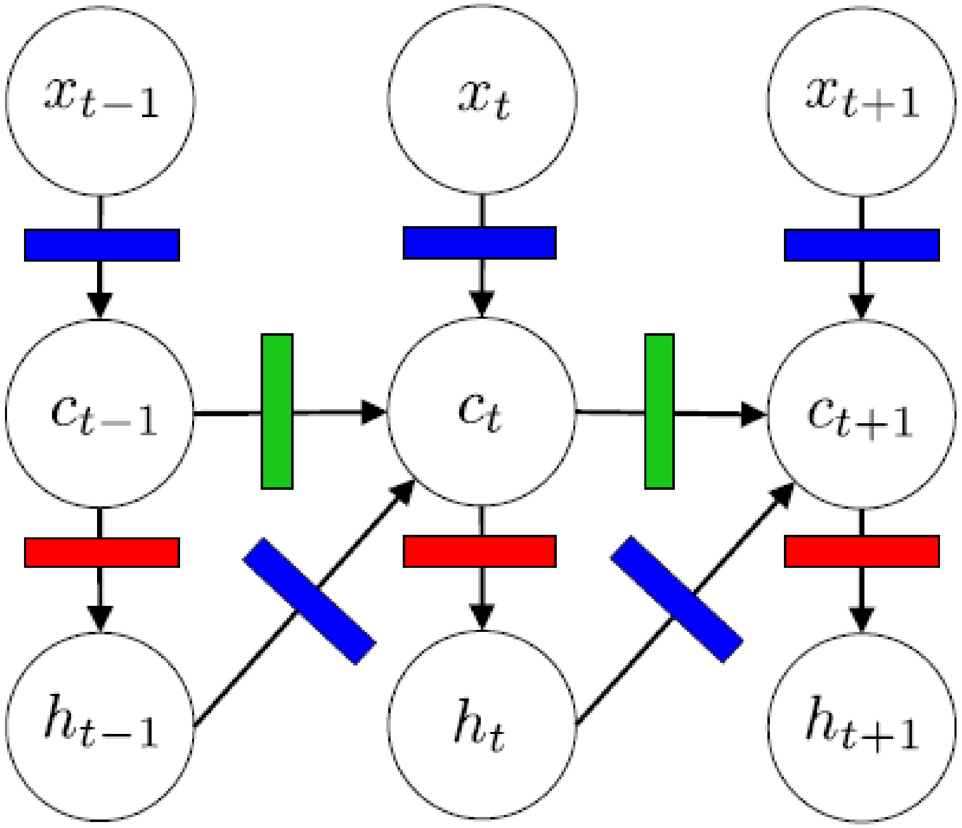
LSTM

- Long Short-Term Models (LSTM)
  - solve the vanishing gradients problem
  - memory cells (a vector) that can preserve gradients across time.
  - gating components, smooth mathematical functions that simulate logical gates.
  - a gate $g \in [0, 1]^n$ is a vector multiplied component-wise with another vector $v \in \mathbb{R}^n$, and the result is then added to another vector. The values of g are designed to be close to either 0 or 1.

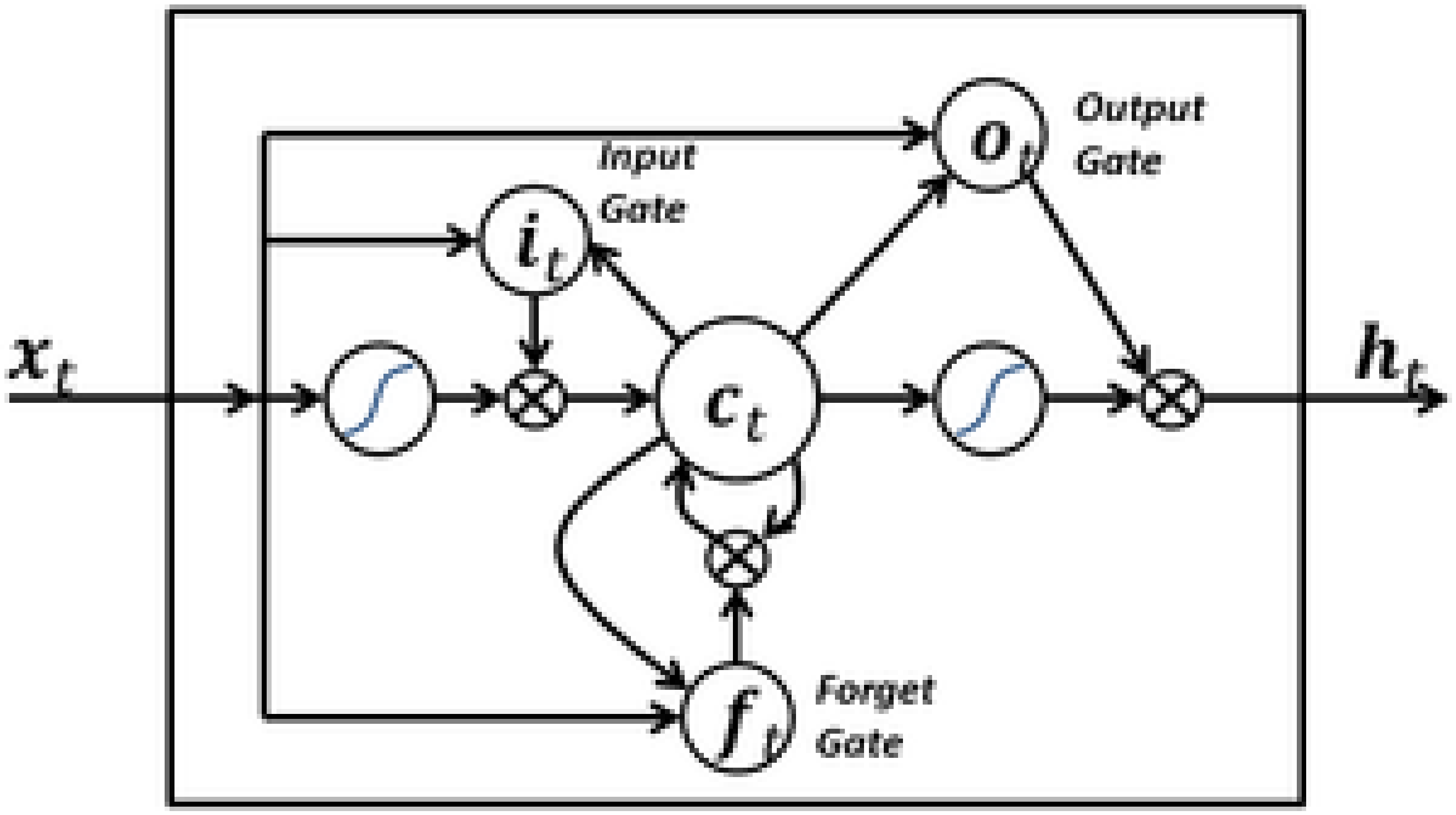
LSTM

$$c_t = f_t c_{t-1} + i_t \tanh\left(W^{(xc)} x_t + W^{(hc)} h_{t-1} + b^{(c)}\right)$$

$h_t = o_t \tanh(c_t)$



LSTM

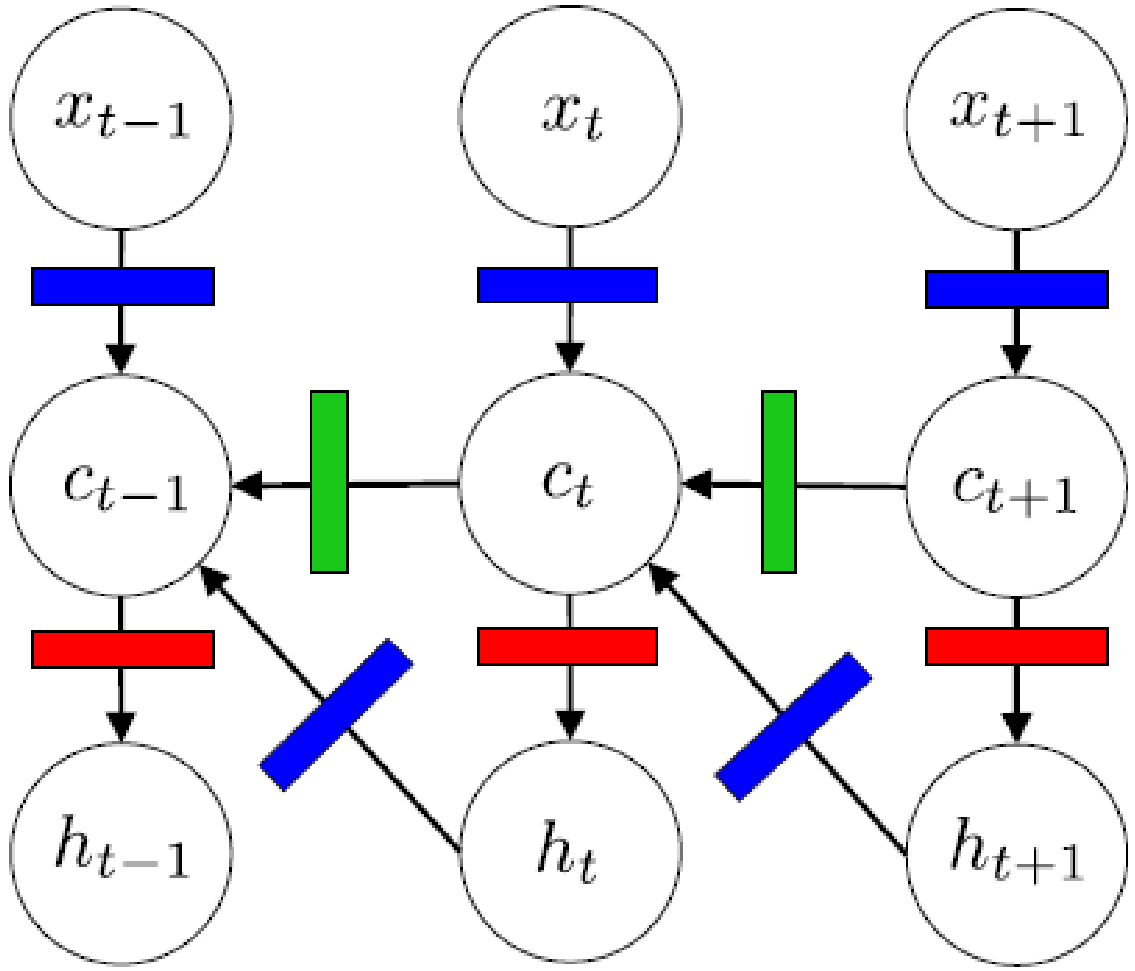


LSTM

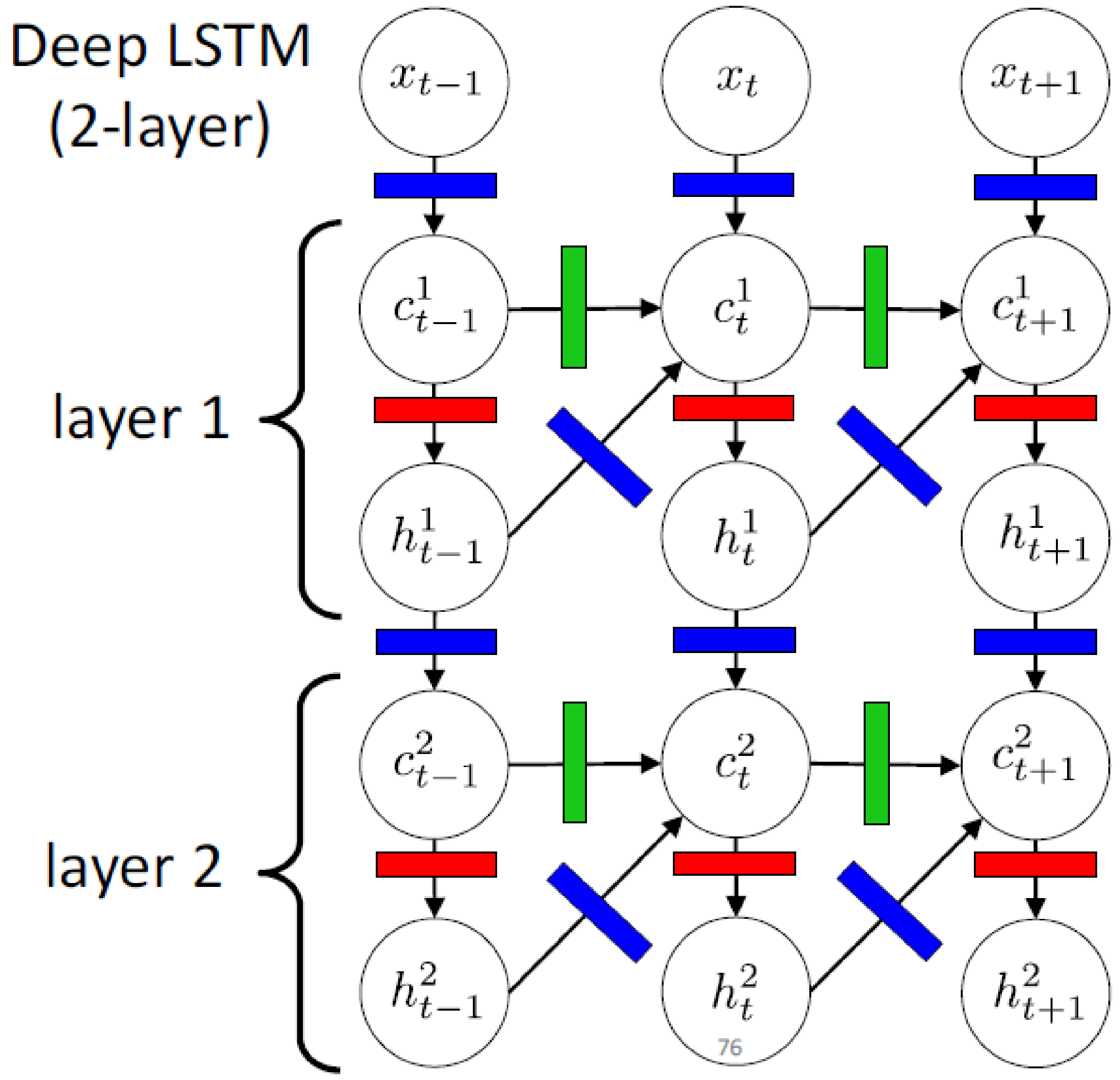
$$
\begin{aligned}
i_t &= \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i\right) \\
f_t &= \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f\right) \\
c_t &= f_t c_{t-1} + i_t \tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right) \\
o_t &= \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o\right) \\
h_t &= o_t \tanh(c_t)
\end{aligned}
$$

LSTM

Bidirectional LSTM



LSTM

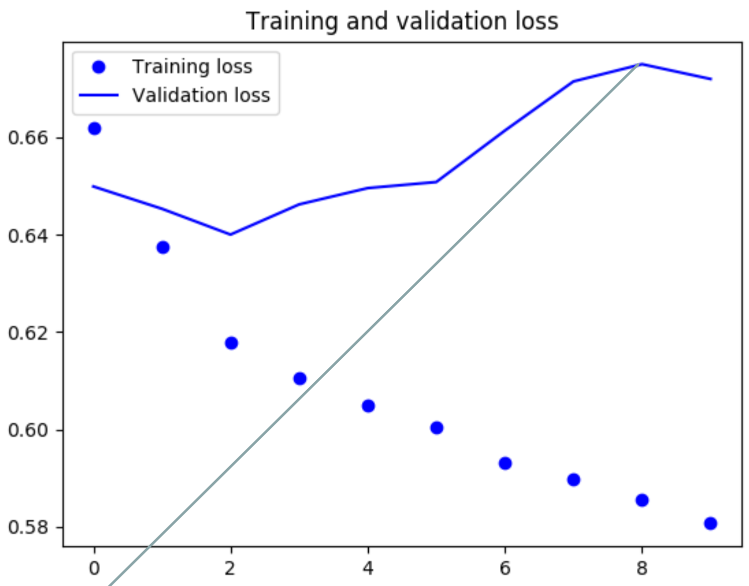

Deep LSTM (2-layer)
layer 1
layer 2
x_{t-1}
x_t
x_{t+1}
c^1_{t-1}
c^1_t
c^1_{t+1}
h^1_{t-1}
h^1_t
h^1_{t+1}
c^2_{t-1}
c^2_t
c^2_{t+1}
h^2_{t-1}
h^2_t
h^2_{t+1}
76

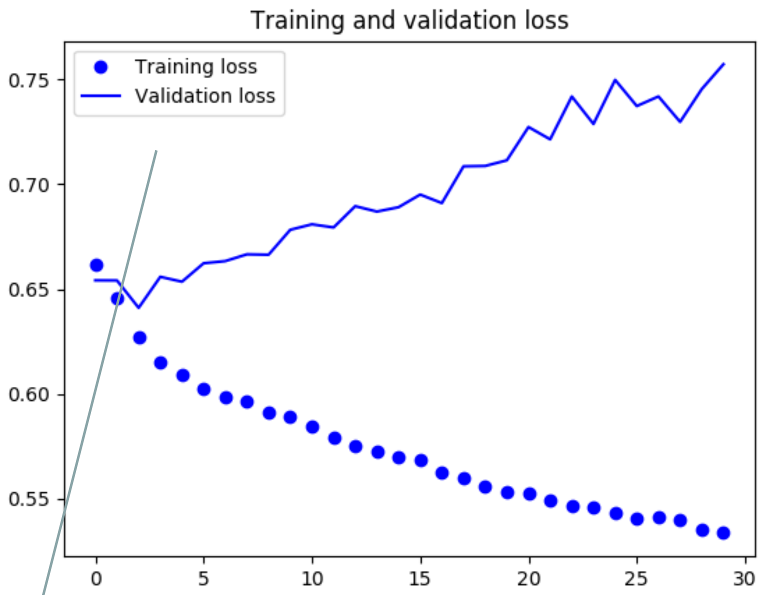
LSTM with Keras

```
from keras.layers import LSTM
```

```
def buildKerasSimpleLSTMModel(parameters, x_train, y_train, x_test, y_test, epochs,
dropout=None):
    global history
    input_shape = (x_train.shape[1],x_train.shape[2]); print 'input_shape', input_shape
    model = Sequential()
    if dropout:
        model.add(LSTM(32, input_shape=input_shape, dropout=dropout,
        recurrent_dropout=dropout))
    else:
        model.add(LSTM(32, input_shape=input_shape))
    model.add(Dense(1, activation='sigmoid'))
    model.summary()
    model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
    history = model.fit(x_train, y_train, epochs=epochs, batch_size=128,
                                validation_split=0.2, verbose=2)
    return model
```

LSTM with Keras





Possible max
10 epochs

Not absolute max
30 epochs
Overfitting

LSTM

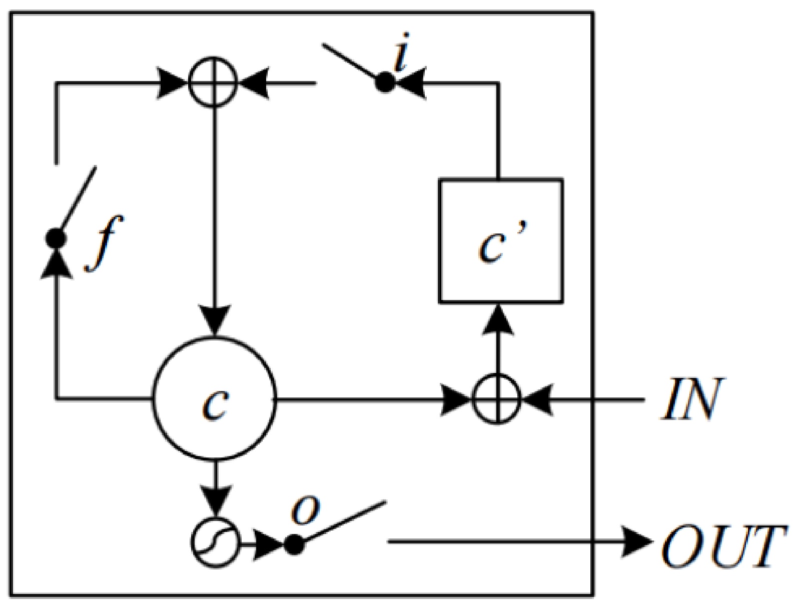
- Facing overfitting in LSTM models
  - Recurrent dropout.
  - Stacking recurrent layers, to increase the representational power of the network.
  - Bidirectional recurrent layers, which presents the same information to a recurrent network in different ways, increasing accuracy and mitigating forgetting issues.

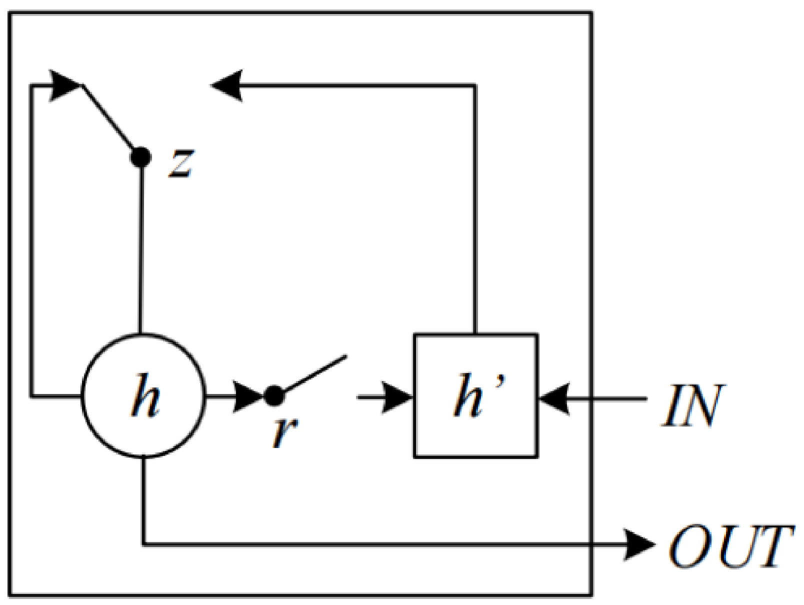
Bidirectional LSTM with Keras

```
def buildKerasBiLSTMModel(parameters, x_train, y_train, x_test, y_test, epochs,
dropout=None):
    global history
    input_shape = (x_train.shape[1],x_train.shape[2]); print 'input_shape', input_shape
    model = Sequential()
    if dropout:
        model.add(layers.Bidirectional(layers.LSTM(32, dropout=dropout,
recurrent_dropout=dropout), input_shape=input_shape))
    else:
        model.add(layers.Bidirectional(layers.LSTM(32),input_shape=input_shape))
    model.add(Dense(1, activation='sigmoid'))
    model.summary()
    model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
    history = model.fit(x_train, y_train, epochs=epochs, batch_size=128, validation_split=0.2,
verbose=2)
    return model
```

LSTM vs GRU

LSTM vs GRU





GRU

- Gated Recurrent Unit
  - Based on the same principles as the LSTM, but uses fewer filters and operations to calculate $h^{(t)}$. Filter update (update gate) $z^{(t)}$. and reset the filter status (reset gate) $r^{(t)}$.
  - cheaper to run, albeit they may not have quite as much representational power as LSTM. Trade-off between computational expensiveness and representational.
  - Calculated using the following formulas:

$$z^{(t)} = \sigma(W^{z}x^{(t)} + U^{z}h^{(t-1)})$$
$$r^{(t)} = \sigma(W^{r}x^{(t)} + U^{r}h^{(t-1)})$$

GRU with Keras

```
def buildKerasSimpleGRUModel(parameters, x_train, y_train, x_test, y_test, epochs,
dropout=None):
    global history
    input_shape = (x_train.shape[1],x_train.shape[2]); print 'input_shape', input_shape
    model = Sequential()
    if dropout:
        model.add(layers.GRU(32, input_shape=input_shape, dropout=dropout,
recurrent_dropout=dropout))
    else:
        model.add(layers.GRU(32, input_shape=input_shape))
    model.add(Dense(1, activation='sigmoid'))
    model.summary()
    model.compile(optimizer='rmsprop', loss='mae', metrics=['acc'])
    history = model.fit(x_train, y_train, epochs=epochs, batch_size=128, validation_split=0.2,
verbose=2)
    return model
```

Stacked GRU with Keras

```
def buildKeras2LGRUModel(parameters, x_train, y_train, x_test, y_test, epochs,
dropout=None):
    global history
    input_shape = (x_train.shape[1],x_train.shape[2]); print 'input_shape', input_shape
    model = Sequential()
    if dropout:
        model.add(layers.GRU(32, input_shape=input_shape, return_sequences=True,
dropout=dropout[0], recurrent_dropout=dropout[1]))
        model.add(layers.GRU(64, activation='relu', dropout=dropout[0],
recurrent_dropout=dropout[1]))
    else:
        model.add(layers.GRU(32, return_sequences=True, input_shape=input_shape))
        model.add(layers.GRU(64, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.summary()
    model.compile(optimizer='rmsprop', loss='mae', metrics=['acc'])
    history = model.fit(x_train, y_train, epochs=epochs, batch_size=128,
validation_split=0.2, verbose=2)
    return model
```

NN models for NLP

- Feed-forward neural Networks (FFNM)
- Convolutional NN (CNN)
- Recurrent NN (RNN)
- Autoencoders
- Recursive NN (RecNN)
- Gated models
  - Long Short Time Models (LSTM)
  - Gated Recurrent Units (GRU)
- Attention-based models

Attention-based models

- One potential problem that the traditional encoder-decoder framework faces is that the encoder at times is forced to encode information which might not be fully relevant to the task at hand. The problem arises also if the input is long or very information-rich and selective encoding is not possible.
- For example, the task of text summarization can be cast as a **sequence-to-sequence** learning problem, where the input is the original text and the output is the condensed version. Intuitively, it is unrealistic to expect a fixed-size vector to encode all information in a piece of text whose length can potentially be very long. Similar problems have also been reported in machine translation.
- In tasks such as text summarization and machine translation, certain alignment exists between the input text and the output text, which means that each token generation step is highly related to a certain part of the input text. This intuition inspires the **attention mechanism**.

Attention-based models

- This mechanism attempts to ease the above problems by allowing the decoder to refer back to the input sequence. Specifically during decoding, in addition to the last hidden state and generated token, the decoder is also conditioned on a “context” vector calculated based on the input hidden state sequence.
- [Bahdanau et al. 2014], [Luong et al, 2016], MT
- [Rush et al. 2015] Summ
- In aspect-based sentiment analysis, [Wang et al. 2016] proposed an attention-based solution where they used aspect embeddings to provide additional support during classification. The attention module focused on selective regions of the sentence which affected the aspect to be classified.

Attention-based models

from Luong et al, 2016: Conventional neural machine translation (NMT)

A basic form of NMT consists of two components:

(a) an encoder which computes a representation s for each source sentence and

(b) a decoder which generates one target word at a time and hence decomposes the conditional probability as:

$$\log p(y|x) = \sum_{j=1}^{m} \log p\left(y_j | y_{<j}, s\right)$$

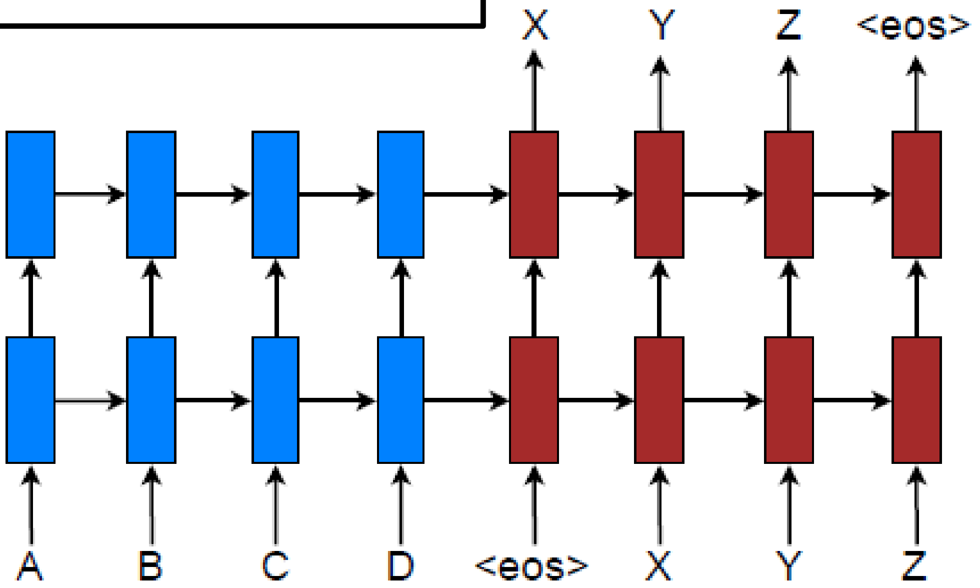


Figure 1: **Neural machine translation** – a stacking recurrent architecture for translating a source sequence A B C D into a target sequence X Y Z. Here, <eos> marks the end of a sentence.

Attention-based models

- In Luang et al, 2016 attention-based models are classified into two broad categories, global and local. These classes differ in terms of whether the “attention” is placed on all source positions or on only a few source positions.

Attention-based models

from Luong et al, 2016:

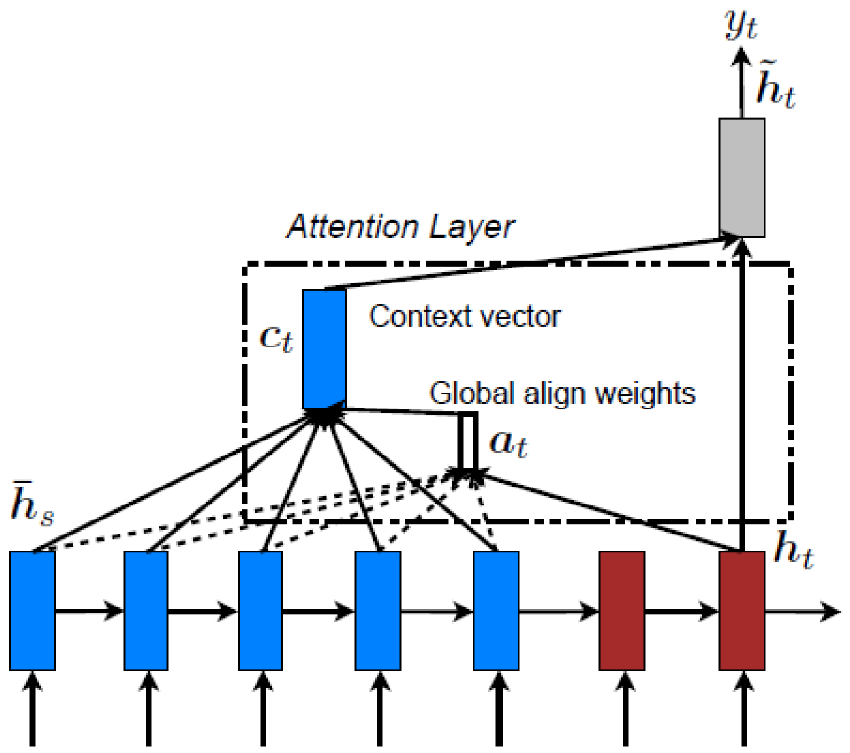


Figure 2: **Global attentional model** – at each time step t, the model infers a *variable-length* alignment weight vector a_t based on the current target state h_t and all source states $\bar{h}_s$. A global context vector c_t is then computed as the weighted average, according to a_t, over all the source states.

Attention-based models

from Luong et al, 2016:

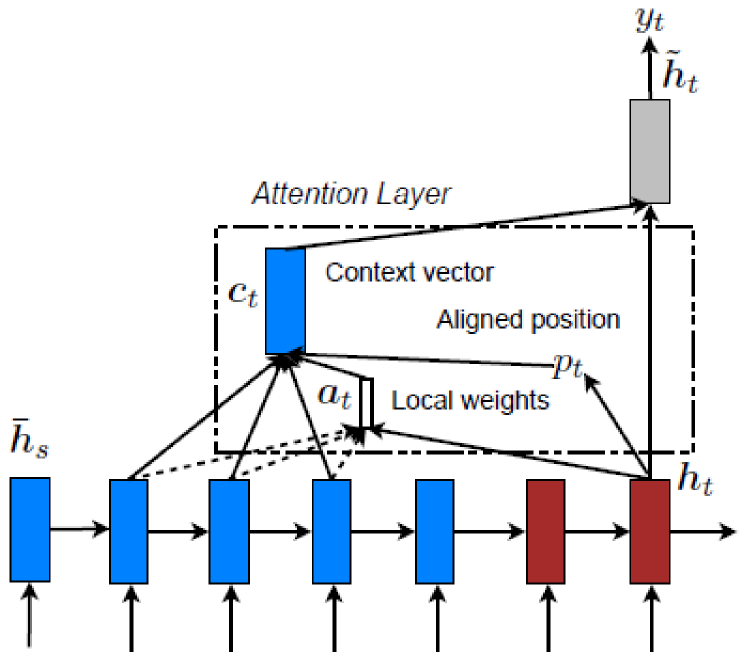


Figure 3: **Local attention model** – the model first predicts a single aligned position p_t for the current target word. A window centered around the source position p_t is then used to compute a context vector c_t, a weighted average of the source hidden states in the window. The weights a_t are inferred from the current target state h_t and those source states $\bar{h}_s$ in the window.

Attention-based models

- These attentional decisions are made independently, which is suboptimal. Whereas, in standard MT, a coverage set is often maintained during the translation process to keep track of which source words have been translated.
- Likewise, in attentional NMTs, alignment decisions should be made jointly taking into account past alignment information. To address that, we propose an input-feeding approach in which attentional vectors $\tilde{h}_t$ are concatenated with inputs at the next time steps

Attention-based models

from Luong et al, 2016:

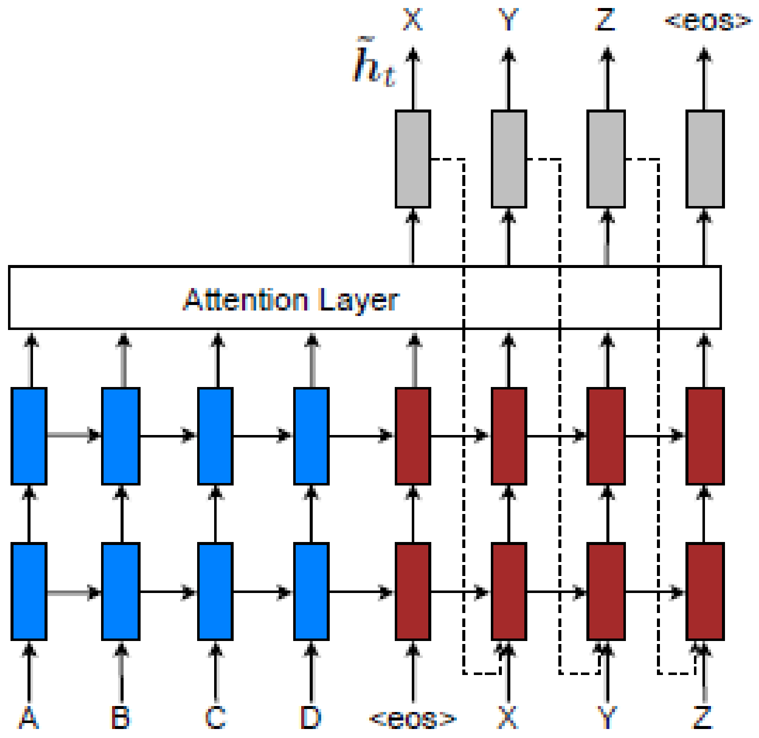


Figure 4: **Input-feeding approach** – Attentional vectors $\tilde{h}_t$ are fed as inputs to the next time steps to inform the model about past alignment decisions.