UE 803 - Data Science for NLP

Lecture 14: Neural Classification

Claire Gardent - CNRS / LORIA

Loria Laboratoire lorrain de recherche en informatique et ses applications

UNIVERSITÉ DE LORRAINE

Outline

A Short Introduction to Neural Networks

- Perceptron vs Neural Network
- Computing Activation Values
- Network architectures
- Training
  - Back Propagation
  - Stochastic Gradient Descent

Classifying text

- with Multi-Layer Perceptron
- with Convolutional Neural Networks
- with Recurrent Neural Networks

Perceptron vs. Neural Network

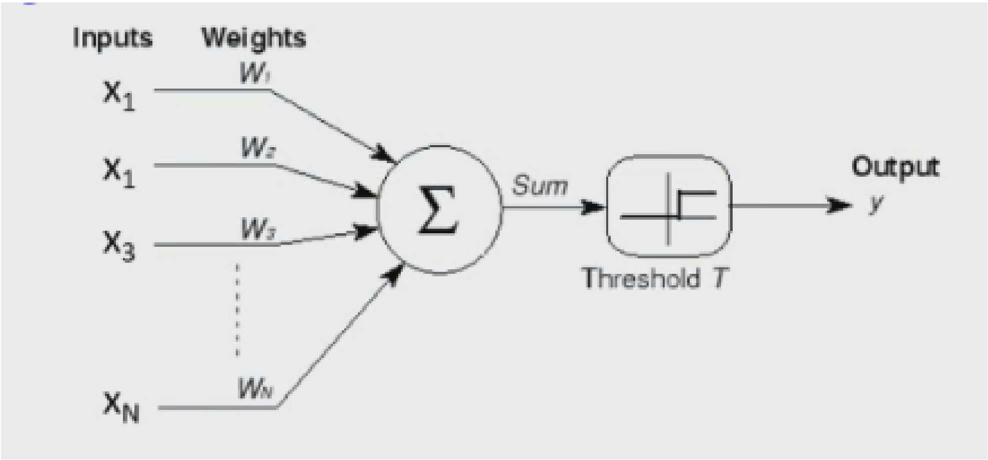
Remember the Perceptron ?

Takes as input x with features x_i and the current parameter values w_i,

computes the weighted sum of the features

$$A_w(x) = \sum_i w_i \times x_i$$

If $A_w(x)) \geq 0$:
then the output is 1 (spam)
else the output is -1 (not spam)



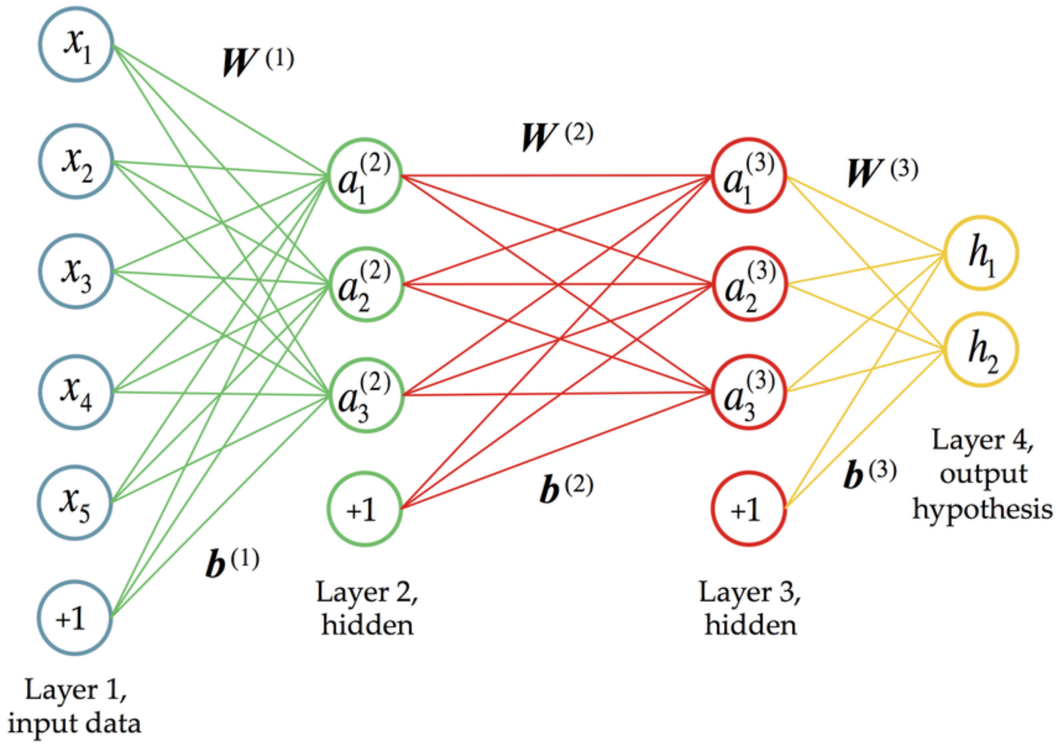
Example Perceptron Output

x	$f(x)$	w	$\sum_i w_i \cdot f_i(x)$
"free money"	BIAS : 1	BIAS : -3	$(1)(-3)\ +$
	free : 1	free : 4	$(1)(4)\ +$
	money : 1	money : 2	$(1)(2)\ +$
	the : 0	the : 0	$(0)(0)\ +$
	...	...	...
			$= 3$

$A_w(x) = 3 \geq 0$

So the output is 1 and the mail ("free money") is classified as spam.

Neural Network



- A neural network is a collection of basic elements, called neurons or (somewhat misleadingly) perceptrons
- Each neuron produces an ***activation value*** which is passed on as input (signal) to other neurons , thus producing a cascading effect

Computing Activation Values

Activation Values

A neuron applies an ***activation function*** to the ***weighted sum of its inputs*** to returns an ***activation value*** .

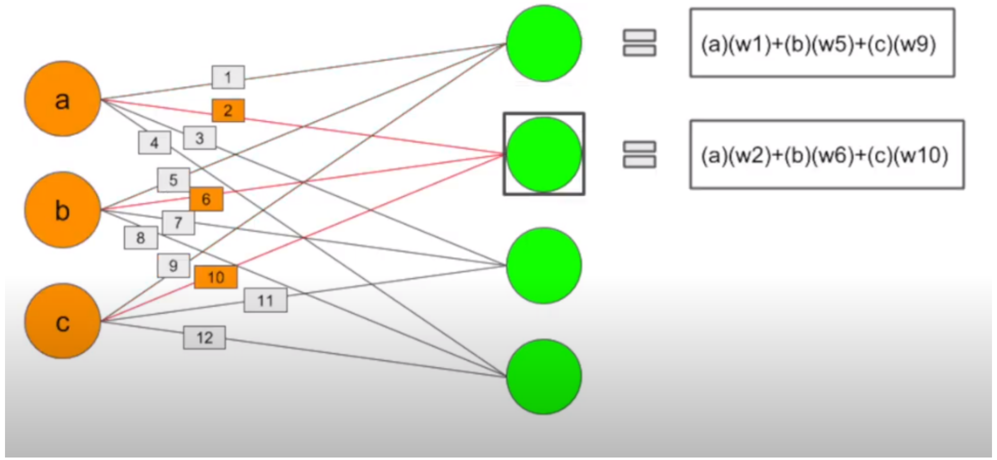
- Each neuron has its own set of weights
- Input values for Neuron 2: a, b, c
- Weighted sum for Neuron 2:

$$w2a + w6b + w10c$$

- Activation value Neuron 2:

$$g(w2a + w6b + w10c)$$

with g some activation function



Forward Computation

Activation values are computed using matrix multiplication

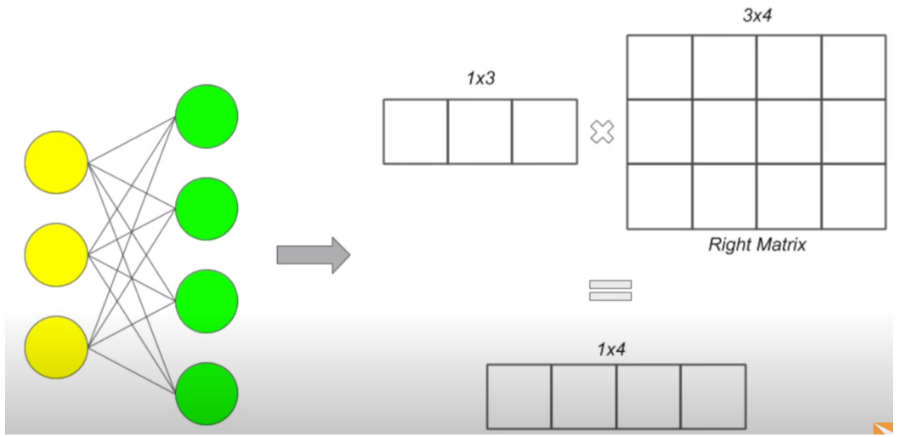
$$OutputAV = Input * WeightMatrix$$

- Inputs Matrix

(1, # Input Neurons)

- Weight Matrix

(# Input Neurons, # Output Neurons)



Output Matrix

(1, # Output Neurons)

Forward Computation

Activation values are computed using matrix multiplication

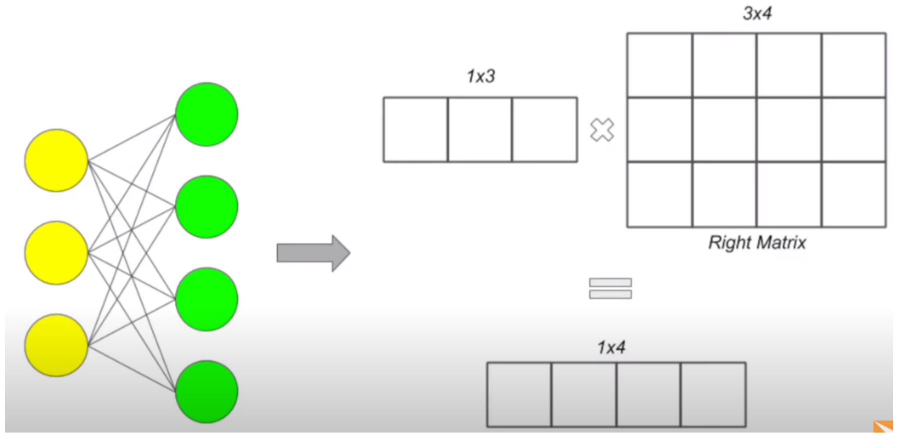
$$Input * WeightMatrix = Output$$

- 3 inputs

$$InputMatrix : (1, 3)$$

- 3 weights for each of the 4 output neurons

$$WeightMatrix : (3, 4)$$



4 output neurons

$$OutputMatrix : (1, 4)$$

Forward Computation

Neuron 1 Activation value

$$g(w1a + w5b + w9c)$$

Neuron 2 Activation value

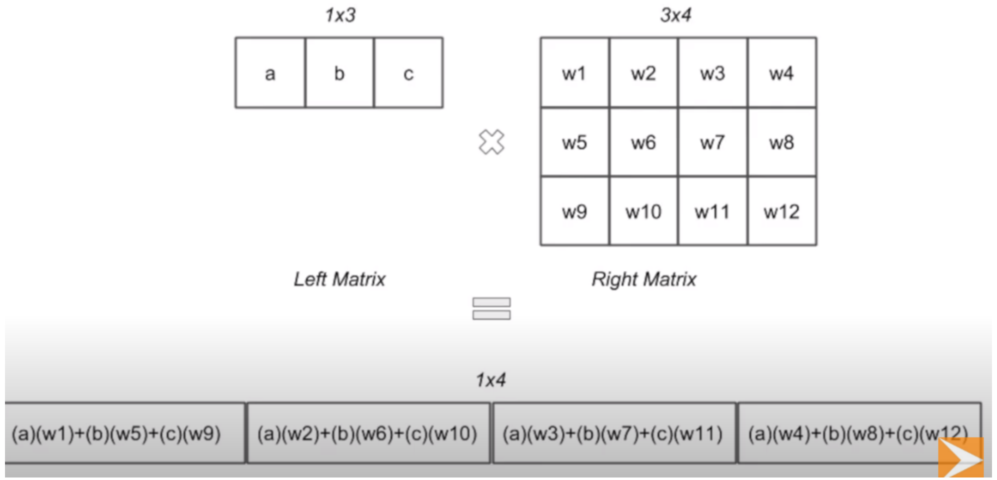
$$g(w2a + w6b + w10c)$$

Neuron 3 Activation value

$$g(w3a + w7b + w11c)$$

Neuron 4 Activation value

$$g(w4a + w8b + w12c)$$



What is g ?

Activation Functions Used in Neural Networks

Output Layer

- Binary classification
  Sigmoid: range = [0,1], binary classification

- N-ary classification
  Softmax: Returns a probability distribution

- Regression
  Hyperbolic Tangent (Tanh): range = (-1,1)

Other layers

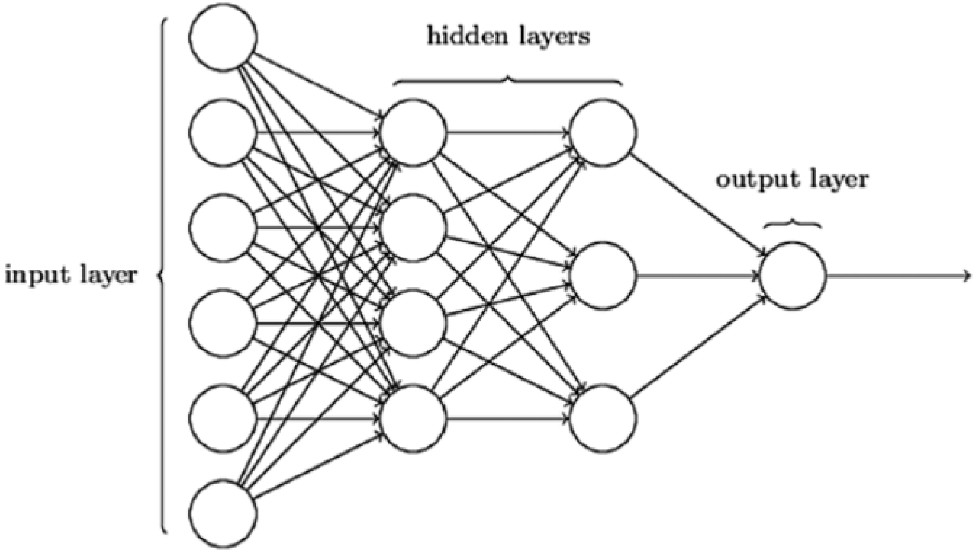
- ReLU: Rectified Linear Unit
- Leaky ReLU
- ELU
- MaxOut

Blog 1, Blog 2

Network Architectures

Neural Network Architecture

- Neurons are organized in **layers**
- The layers between input and output are referred to as **hidden layers**
- Layers are connected
- The density and type of connections between layers is the **configuration**
  E.g., a **fully connected configuration** has all the neurons of layer L connected to those of L + 1.

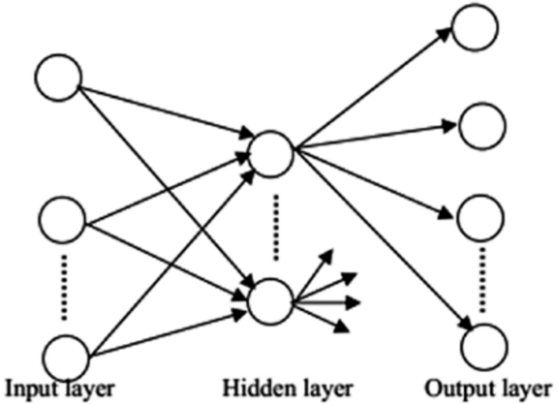


Feedforward Neural Network

- Also called ***Multi-Layer Perceptron***

- ***Fully connected***
  All the neurons of layer L are connected to all neurons of layer L + 1.

- When used for ***classification***
  Size of output layer = Number of classes In the output layer, the activation function is Sigmoid if the classification is binary and softmax if there are more than 2 classes

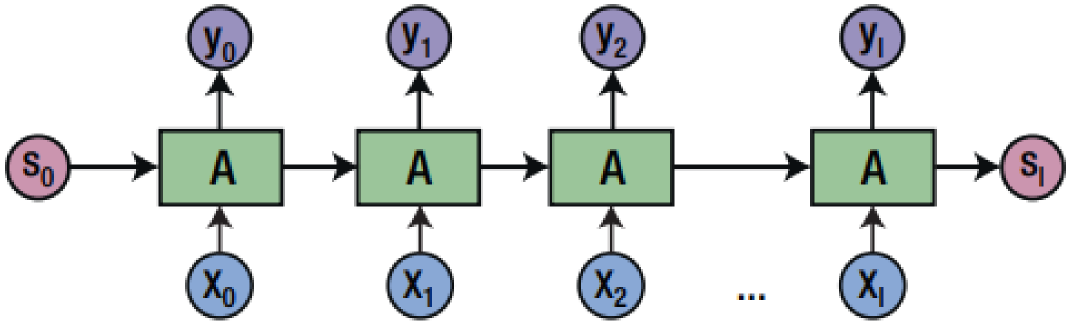


Convolutional Neural Networks

- Well adapted for ***image recognition*** and handwriting recognition.
- Based on sampling a window or portion of an image, detecting its features, and then using the features to build a representation.
- Deep networks (several layers)

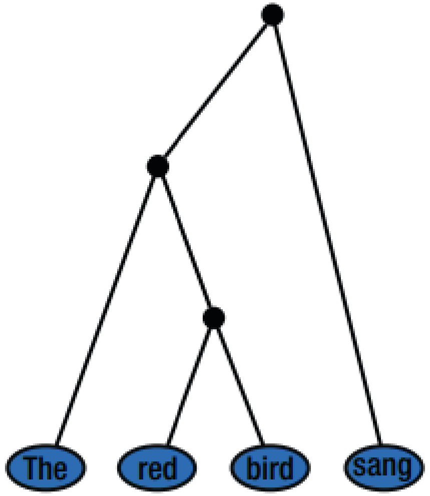
Recurrent Neural Networks

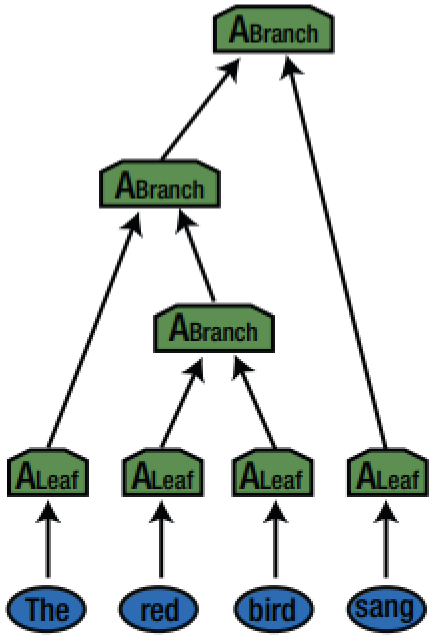
- Well adapted for ***sequences*** of arbitrary length (videos, text)
- Applies the same layer to the input at each time step, using the current input and the hidden state of the previous time steps as inputs



Recursive Neural Networks

- Well adapted for ***tree shaped input*** (e.g., parse trees)
- a fixed set of weights is recursively applied onto the network structure





Training a Neural Network

Learning a NN - The Back-Propagation Algorithm

The weights of a NN are learned by applying the following operation iteratively to each instance in the training data.

Forward Pass
Compute the activation for all neurons and pass them on to the next layer.
The output layer outputs a prediction

Compute the loss
by comparing prediction and expected result
(using cross entropy for classification, mean square error for regression)

Backward Pass
Adjust the NN weights starting with the last layer
This is done by computing the derivative of the loss (the gradient) for each weight

N.B. In practice this is done in batches (not one training instance at a time)

Cost (Loss) Function

- During training, for each training example in the training set (x_i, y_i), the feature vector x_i is input to the neural network, and the network's predicted output $\hat{y_i}$ is compared with the corresponding label y_i.
- The loss function $J(W)$ measures how much the network's predicted output is different than the expected output (the corresponding label).
- The cost function $J(W)$ is a function of the neural network parameters (because $\hat{y}$ is a function of W)

Cost (Loss) Function

For classification problems, the loss function is ***cross entropy*** (also called ***logLoss*** for binary classification)

$$J(W) = -\frac{1}{m}\sum_{j=1}^{m}\sum_{i=1}^{c} y_i^{(j)} ln(y_i^{\hat{(j)}}))$$

For regression tasks, the cost function is the ***quadratic loss function***

$$J(W) = -\frac{1}{2m}\sum_{j=1}^{m} \mid\mid y^{(j)} - y^{\hat{(j)}} \mid\mid^2$$

Example

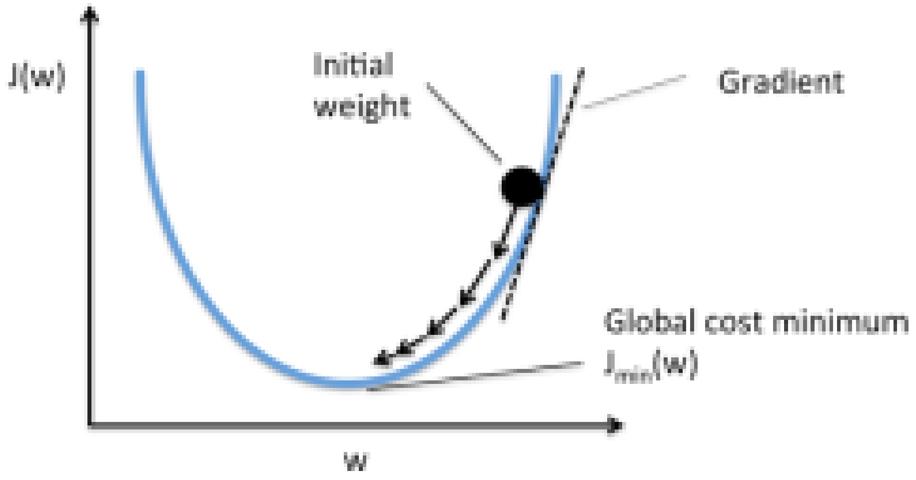
```
import numpy as np

# True labels
# A class is represented by a probability distribution over the classes
# Here Y has probability 1 for class 1 and 0 for the others
Y = np.array([1,0,0])

# Predicted Labels
# Y1 predicts probability 0.7 for class 1, 0.2 for class 2, 0.1 for class 3
Y1 = np.array([0.7,0.2,0.1])
Y2 = np.array([0.1,0.3,0.6])

# Cross entropy loss
l1 = np.sum(-Y * np.log(Y1))
l2 = np.sum(-Y * np.log(Y2))

>>>>
# Y1 is a better prediction (higher probability for class 1, smaller loss)
np.log(Y1): [-0.35667494 -1.60943791 -2.30258509].
np.log(Y2): [-2.30258509 -1.2039728  -0.51082562].
loss 1: 0.35667494393873245.
loss 2: 2.3025850929940455.
```

Backward Pass

Stochastic Gradient Descent (SGD)

- SGD minimizes the cost function J
- by updating the parameters W of the neural network.

SGD finds the values of the neural network parameters W that minimize the cost function $J(W)$



SGD

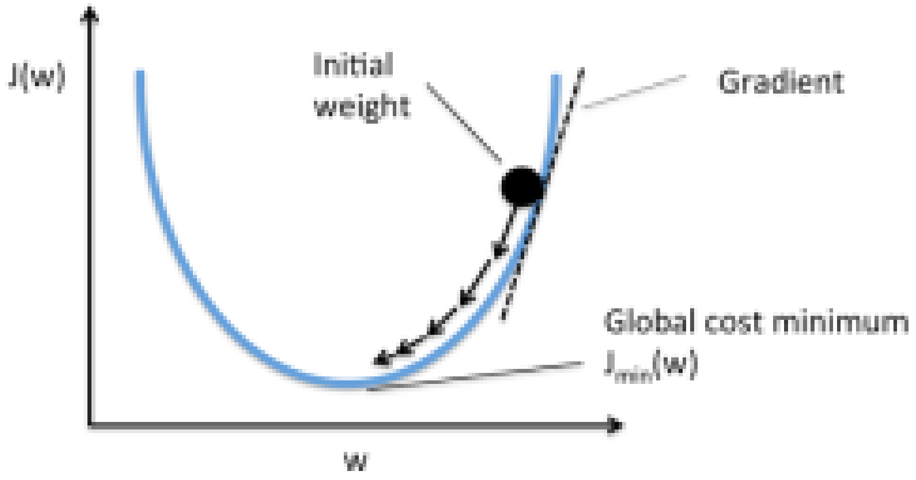
Stochastic Gradient Descent (SGD)

- updates the weights according to following rule (η = learning rate hyperparameter):

$$w \leftarrow w - \eta \frac{dJ(w)}{dw}$$

- moves each weight in the direction of the derivative (***gradient***)

- E.g., on the picture $dJ(w)$ is positive, hence the update rule decreases the value of w and $J(w)$ decreases.



Neural Network Hyper-parameters

The following metrics are used to specify a neural network. They are referred to as hyper parameters.

Architecture

- number and size of layers

Training

- number of iteration over the training corpus
- batch size
- learning rate
- optimizer (algorithm used to adjust the weights)
- loss function (cross entropy, ranking loss etc.)

Classifying Text with Neural Networks

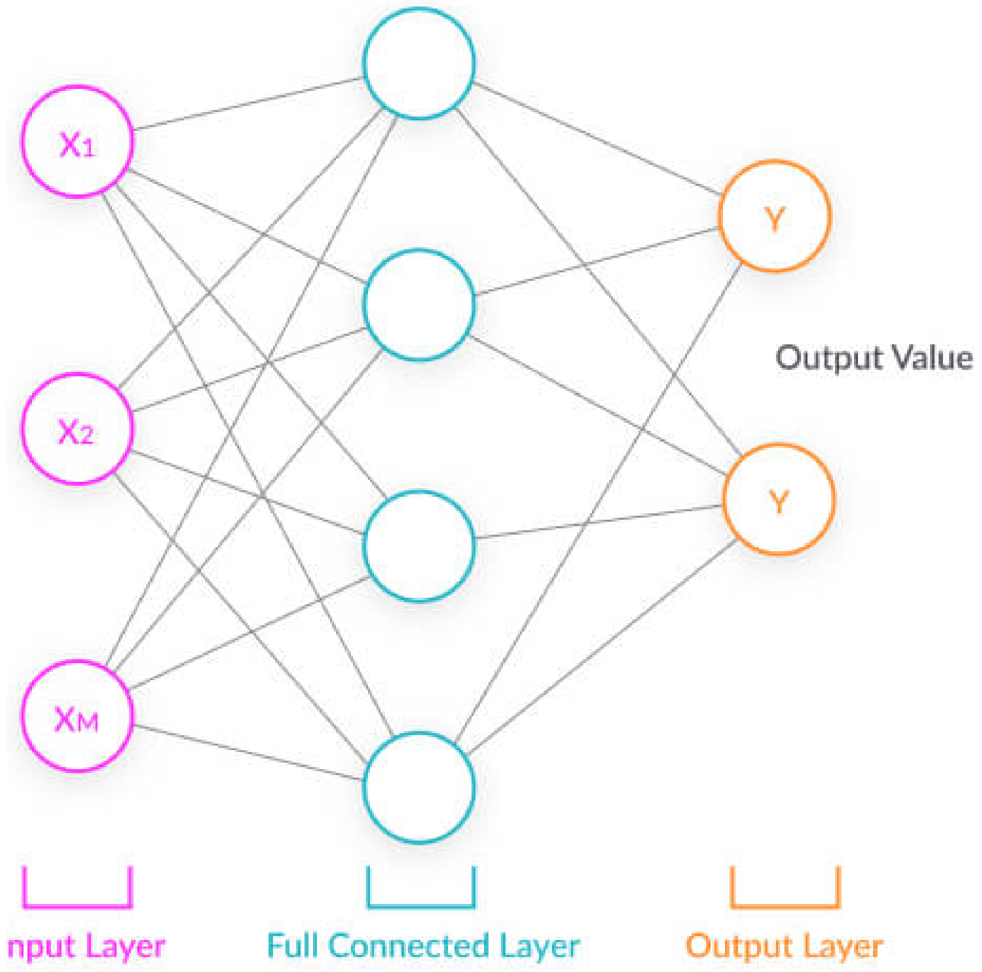
Text Classification using a Multi-Layer Perceptron

- Input layer = Features
  Usually a matrix whose columns are word vectors

- Output layer = class probabilities
  The size of that layer = the number of classes

- One or more hidden layers

- Each node of the hidden layers performs a linear weighting of its inputs from previous layer and passes result through an activation function to nodes in next layer



Output Layer

Regression: single output neuron with no activation function (a score, logit)

Binary classification

- single output neuron with a sigmoid activation function which outputs a value between 0 and 1 (probability of class 1).
- can be turned into a class value by using a threshold and assigning values less than the threshold to 0 otherwise to 1.

Multi-class classification

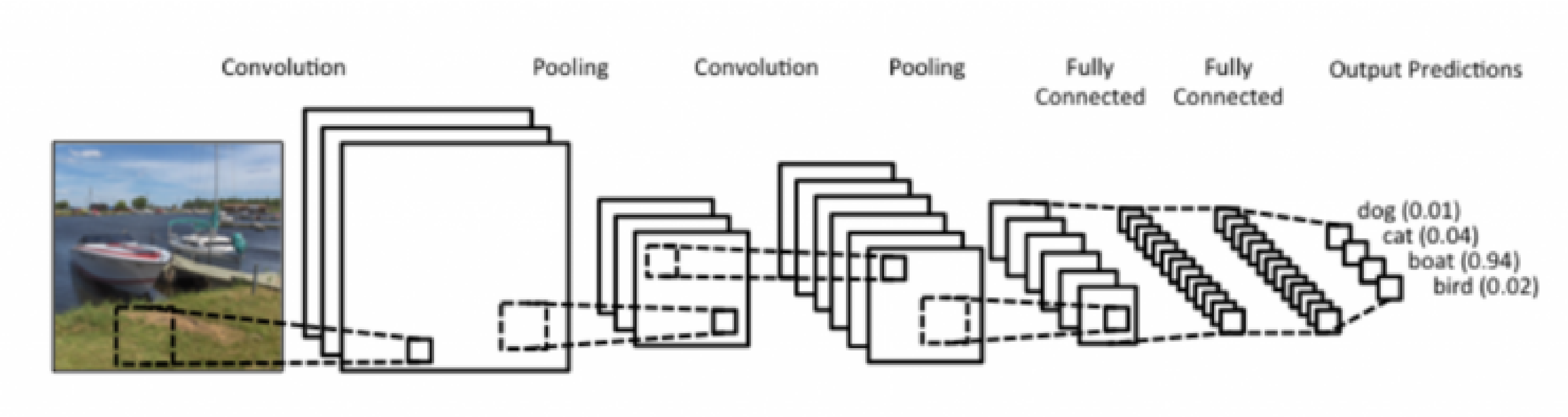
- multiple neurons in the output layer, one for each class
- a softmax activation function may be used to output a probability of the network predicting each of the class values.
- Neuron with the highest probability $\Rightarrow$ class value.

Text Classification with a Convolutional Neural Network (CNN)

- Massively used in Computer Vision. Also used in NLP.
- Compute representations for input subparts
  Extract features of the input
- Less computationally expensive then RNN
  - The computations involved are much lighter than the cell computation involved in LSTMs.
  - There is no temporal dependencies between filters, so they can be applied concurrently
- Can also capture long range dependencies by hierarchically increasing the receptive field.

Text Classification with a Convolutional Neural Network (CNN)

A CNN stacks convolutions (filters), non-linearities and Pooling layers

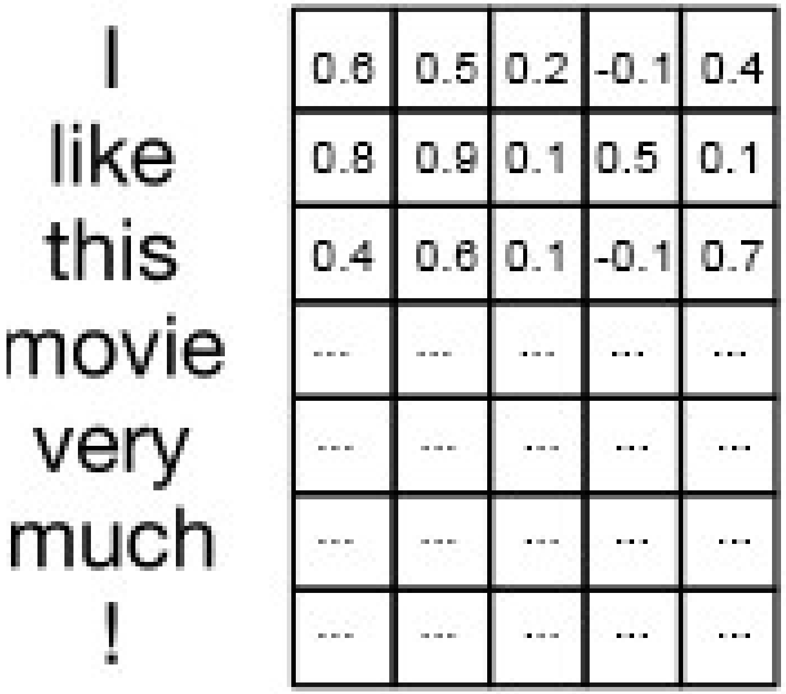


Representing the Input Text

- The input text is converted to a matrix in which each row is a vector (one-hot or embedding) representing a word

- Filters (also called, convolutions or kernels) slide over full rows (words) of the matrix

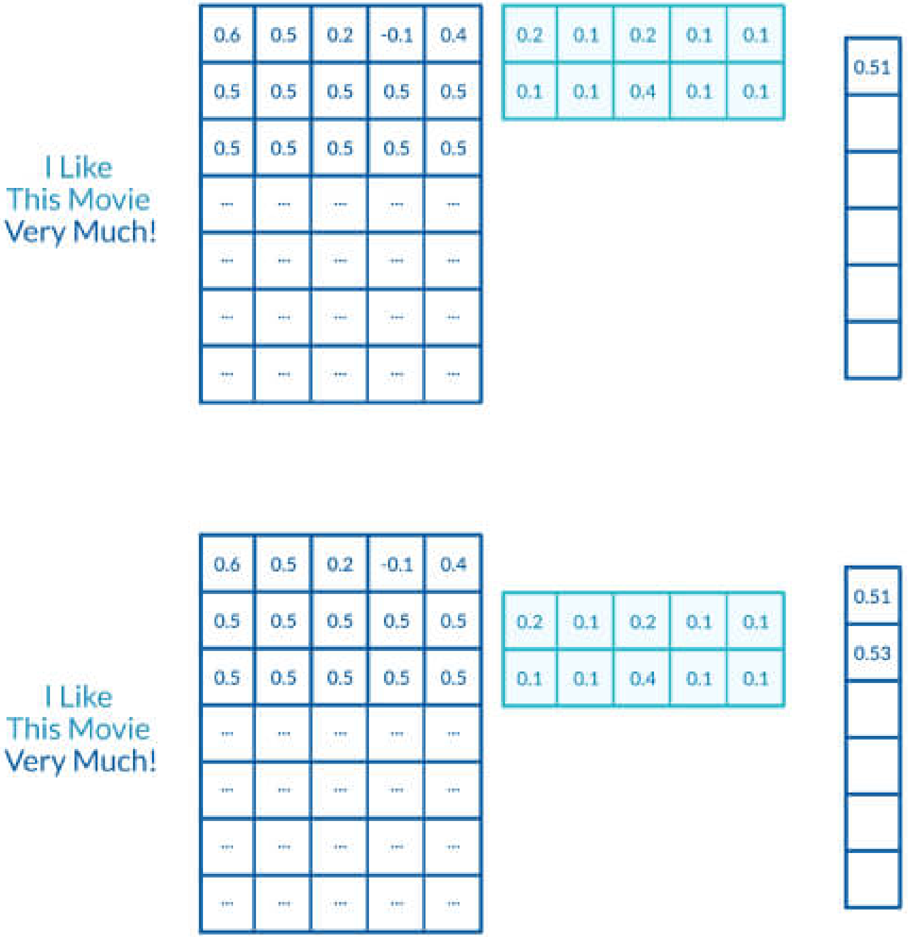
  - hence the width of the filter is the size of the word vectors
  - the height (region size) of the filter varies (2-5 words is common)



Filters CNN (also called Convolutions)

Two-words filter

- slides over a sentence, two words at a time.
- returns a real number (activation value) which is the dot product between the filter and the corresponding input chunk ($W X$)
- The filter is applied over all 2 word sequences in the input
- There can be several filters of various sizes



Filters and Pooling

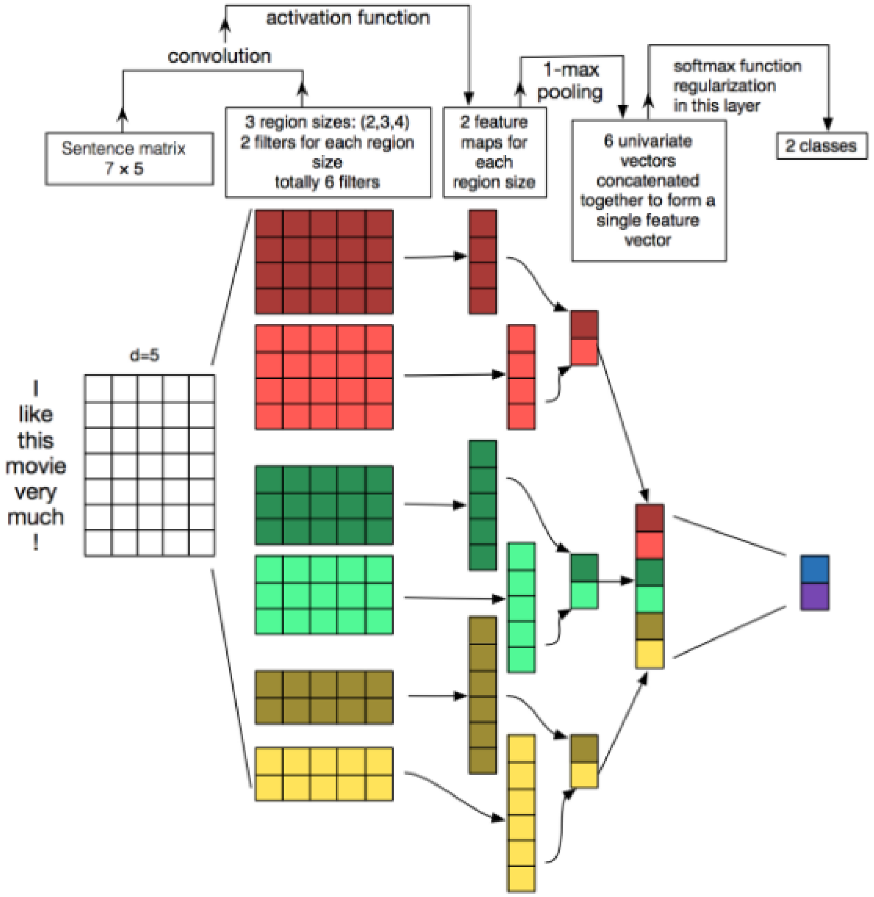
Convolutions

- are applied over all spatial locations in the input
- return an activation value for each location
- The stride determines how convolutions are applied

Pooling

- Usually applies a max operation to the result of each filter.
- in NLP we typically apply pooling over the complete output, yielding just a single number for each filter

Filters and Pooling

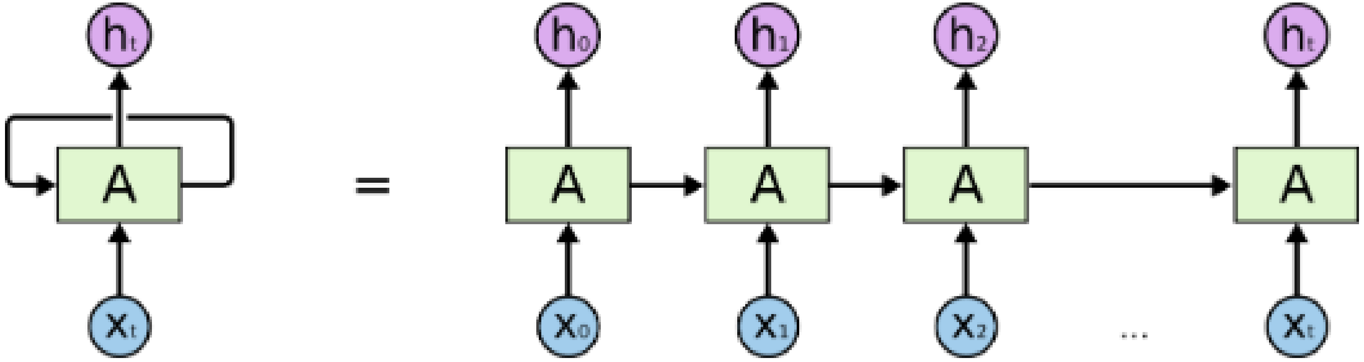


ıage Reference : http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/

Recurrent Neural Network

- can encode sequences of arbitrary length
- takes order into account
- recur over each token in the input sequence and repeatedly perform the same action:

  Given the current input token and the previous hidden state, output a new hidden state



Classifying Text with Pytorch

Text Classification with PyTorch

- Install pytorch
- Load data into tensors
- Define the model
- Define functions to train the model and evaluate results.
- Split the dataset and run the model
- Evaluate the model on test data

PyTorch

- Install pytorch

https://pytorch.org/get-started/locally/

```
conda install pytorch-cpu torchvision-cpu -c pytorch
```

- PyTorch is an open source machine learning framework

- The torchtext (torchvision) package consists of data processing utilities and popular datasets for natural language (vision).

Loading the data into Tensors

```
# Create a tensor for the input data
# the long attribute indicates that integers will be used
X = torch.zeros(num_texts, max_sentence_len).long()

# Populate the tensor with the input data (here the list of labels i.e., Y)
Y = torch.LongTensor(labels)
```

- Tensors are a specialized data structure that are very similar to arrays and matrices.
- Tensors are used to encode the inputs, the parameters and the outputs of a model.
- Tensors are similar to NumPy's ndarrays, except that tensors can run on GPUs or other hardware accelerators.
- Tensors are also optimized for automatic differentiation

Operations on tensors

Over 100 tensor operations, including arithmetic, linear algebra, matrix manipulation (transposing, indexing, slicing), sampling and more are available.

Each of these operations can be run on the GPU (at typically higher speeds than on a CPU). If you're using Colab, allocate a GPU by going to Runtime > Change runtime type > GPU.

By default, tensors are created on the CPU. We need to explicitly move tensors to the GPU using .to method (after checking for GPU availability). Keep in mind that copying large tensors across devices can be expensive in terms of time and memory!

```
### To check whether GPU is available
torch.cuda.is_available()

# To move a tensor to the GPU if available
if torch.cuda.is_available():
   tensor = tensor.to("cuda")

# To switch between gpu and cpu depending on what is available
```

Useful to know Operations

Squeezing and unsqueezing

Squeezing a tensor removes the dimensions or axes that have a length of one.

Unsqueezing a tensor adds a dimension with a length of one.

```
In [89]: t = torch.zeros(5, 3, dtype=torch.long)
         print(t.size())

torch.Size([5, 3])

In [91]: t1 = t.unsqueeze(1)
         print(t1.size())

torch.Size([5, 1, 3])

In [92]: t2 = t1.squeeze()
         print(t2.size())

torch.Size([5, 3])

In [93]: t4 = t.unsqueeze(0)
         print(t4.size())

torch.Size([1, 5, 3])
```

Defining the network

Three main steps

Step 1: define your network as a subclass of either the nn.Module or a subclass of this module

Step 2: specify the architecture of your network (size and number of layers)
Define the init function

Step 3: say how they connect (for each layer, specify input size, output size and activation function)
Define the forward function

MLP Example: Step 1 and 2

Implementing a multi-layer perceptron with two layers.
The input is a text so the size of the input layer is the maximum length of the texts in our dataset.
The size of the output layer is the number of classes the classifier is handling.

```
# Our MultilayerPerceptron is a subclass of the nn.Module Class
class MultilayerPerceptron(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(MultilayerPerceptron, self).__init__()

        # The network consists of two fully connected layers
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, num_classes)
```

MLP Example: Step 3

```
# Specifying how the layers connect
def forward(self, x_in, apply_softmax=False):
        """The forward pass of the MLP

        Args:
            x_in (torch.Tensor): an input data tensor
            apply_softmax=false because we use the cross-entropy loss
        Returns:
            the resulting tensor
        """
        # Apply non linear Relu activation function to the output of the first fully connected layer
        intermediate = F.relu(self.fc1(x_in))
        # Apply non linear Relu activation function to the output of the second fully connected layer
        output = self.fc2(intermediate)

        return output
```

Creating the model

```
# Size of the input layer (max length of the input text)
input_size = 471
# Size of the hidden layers
hidden_size = 128
# Size of the output layer (classification layer)
num_classes = 5

mlp = MultilayerPerceptron(input_size, hidden_size, num_classes)
```

Evaluating the model

```
 def perf(model, loader):
# define the loss
    criterion = nn.CrossEntropyLoss()
# No drop out
    model.eval()
    total_loss = correct = num = 0
    for x, y in loader:
# No gradient computation, weights remain unchanged
      with torch.no_grad():
# Compute the scores for the instances in the input batch
        y_scores = model(x)
# Compute the loss
        loss = criterion(y_scores, y)
# Compute the predictions
        y_pred = torch.max(y_scores, 1)[1]
# Update the batch loss
        total_loss += loss.item()
        num += len(y)
    return total_loss / num
```

Defining the Training Loop

```
def fit(model, epochs):
    # Specify the loss function  and the optimizer
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters())

    # Iterate over epochs (i.e., slices of the data)
    for epoch in range(epochs):
        model.train()                # Set the module in training mode
        total_loss = num = 0         # Initialise the loss to 0
        # Iterate over batches of (x,y) pairs in the training data
        for x, y in train_loader:

            optimizer.zero_grad()    # null the gradients

            y_scores = model(x)      # predict labels for the batch

            loss = criterion(y_scores, y)  # calculate the loss

            loss.backward()          # Back propagate

            optimizer.step()         # Adjust the weights

        print(epoch, *perf(model, train_loader))
```

Training

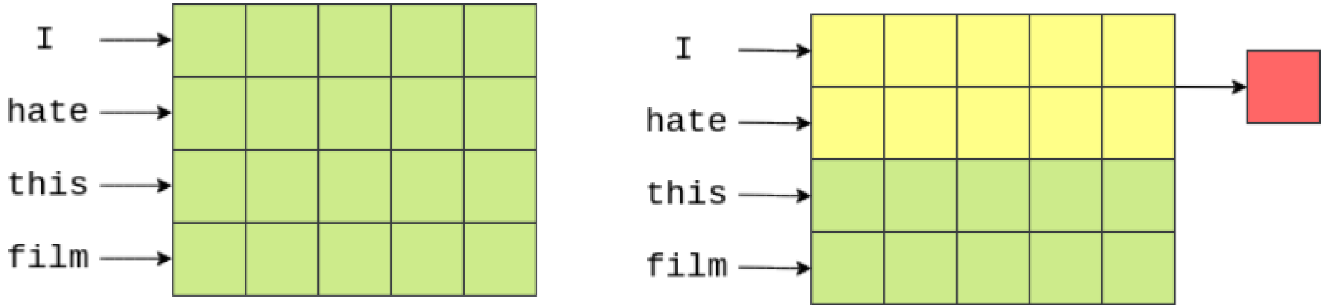
```
fit(mlp, 10)
```

Predicting

```
from sklearn.metrics import confusion_matrix

def predict(model, loader):
    # No drop out
    model.eval()
    correct = 0
    gold = outputs = []
    for x, y in loader:
    # No gradient computation, weights remain unchanged
    with torch.no_grad():
        # Compute the scores for the instances in the input batch
        y_scores = model(x)
        # Compute the predictions
        # y_scores = matrix
        # max(y_scores,1) = max value on lines
        # max(y_scores,1)[1] = index of max value on lines
        y_preds = torch.max(y_scores, 1)[1]
        # Store the gold value
        gold = gold+y.tolist()
        # Store the predicted value
        outputs = outputs+y_preds.tolist()
    print(confusion_matrix(gold,outputs))

predict(rnn_model,valid_loader)
```

Defining a CNN

Example Filter



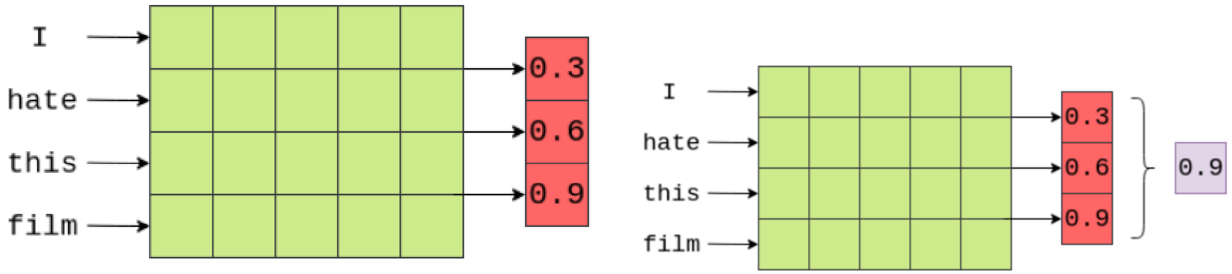
Sentence: (4,5) tensor

- 4 = nb of tokens, 5 = the embedding size

Filter: (2,5) tensor.

- The filter covers two words at a time.
- Each element of the filter has a weight associated with it.
- The output of this filter (shown in red) is a real number that is the weighted sum of all elements covered by the filter

Filter output and Max Pooling



- The filter "scans" the input: it is applied as many time as is possible given the length of the input (and the step, here step = 1)
- A max pooling layer then select the maximum value from the values output by the filter (convolutional layer)
- The output of all max pooling layers (if there are several filters) is concatenated into a single vector and passed through a linear layer to predict the output class.

CNN in pyTorch

100 filters of size 3, 4 and 5

```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 100
N_FILTERS = 100
FILTER_SIZES = [3,4,5]
OUTPUT_DIM = 1
DROPOUT = 0.5

model = CNN(INPUT_DIM, EMBEDDING_DIM, N_FILTERS, FILTER_SIZES, OUTPUT_DIM, DROPOUT)
```

A CNN with three filters of size 3, 4 and 5

```
import torch.nn as nn
import torch.nn.functional as F

class CNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, n_filters, filter_sizes, output_dim,
                 dropout, pad_idx):

        super().__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx = pad_idx)

        self.conv_0 = nn.Conv2d(in_channels = 1,
                                out_channels = n_filters,
                                kernel_size = (filter_sizes[0], embedding_dim))

        self.conv_1 = nn.Conv2d(in_channels = 1,
                                out_channels = n_filters,
                                kernel_size = (filter_sizes[1], embedding_dim))

        self.conv_2 = nn.Conv2d(in_channels = 1,
                                out_channels = n_filters,
                                kernel_size = (filter_sizes[2], embedding_dim))

        self.fc = nn.Linear(len(filter_sizes) * n_filters, output_dim)

        self.dropout = nn.Dropout(dropout)
```

CNN in pyTorch

```
self.conv_0 = nn.Conv2d(in_channels = 1,
                                out_channels = n_filters,
                                kernel_size = (filter_sizes[0], embedding_dim))
```

- The `in_channels` argument is 3 (one channel for each of the red, blue and green channels) for images, 1 for text).
- The `out_channels` is the number of filters (here, 100)
- `kernel_size` the size of the filters

Fixed size output with Max Pooling

- The size of the output of the convolutional layer is dependent on the size of its input

- The max pooling layer ensures that the output of a filter is always 1.

The Forward Function

```
def forward(self, text):
        #text = [batch size, sent len]
        embedded = self.embedding(text)

        #embedded = [batch size, sent len, emb dim]
        #unsqueeze to create the channel dimension expected by the filters
        embedded = embedded.unsqueeze(1)

        #embedded = [batch size, 1, sent len, emb dim]
        conved_0 = F.relu(self.conv_0(embedded).squeeze(3))
        conved_1 = F.relu(self.conv_1(embedded).squeeze(3))
        conved_2 = F.relu(self.conv_2(embedded).squeeze(3))

        #conved_n = [batch size, n_filters, sent len - filter_sizes[n] + 1]
        pooled_0 = F.max_pool1d(conved_0, conved_0.shape[2]).squeeze(2)
        pooled_1 = F.max_pool1d(conved_1, conved_1.shape[2]).squeeze(2)
        pooled_2 = F.max_pool1d(conved_2, conved_2.shape[2]).squeeze(2)

        #pooled_n = [batch size, n_filters]
        cat = self.dropout(torch.cat((pooled_0, pooled_1, pooled_2), dim = 1))

        #cat = [batch size, n_filters * len(filter_sizes)]
        return self.fc(cat)
```

Lab Session

Use a Recurrent Neural Network to classify BBC news articles into 5 topics. The dataset consists of 2225 documents and 5 categories: business, entertainment, politics, sport, and technology.

The exercises cover the following points:

- Converting the text in the corpus to vectors of integers (each integer represents a word in the corpus vocabulary)
- Computing some descriptive statistics to identify a sentence length cutoff (sentences with longer lengths will not be considered for training)
- Specifying, training and testing a recurrent neural network

Useful Links

- Short introduction to Neural NLP
- Video explaining matrix notation for forward computation
- Video with detailed explanation of RNN Pytorch implementation
- Blog on Gradient Descent
- Blog on Advanced Gradient Descent Optimisation
- Blog on CNN for Text Classification
- Another blog on CNN for Text Classification CNN and RNN
- Neural text classification using fasttext
- A detailed explanation of backpropagation