Introduction to Neural Networks:

Understanding Recurrent Neural Networks

By Abhiram Siddanthi and Sreelekshmi Kishore

Neural Networks

There are two fundamentally different approaches to getting a program to do what you want. One is hard coded coding, where you tell the program-specific rules and what outcomes you want, and you guide it throughout the whole way and account for all the possible options the program has to deal with.
On the other hand, you have neural networks where you create a facility for the program to understand what it needs to do on its own. You make this neural network where you provide inputs, tell it what you want as outputs, and then let it figure everything out independently.

For instance, how do you distinguish between a dog and a cat? For one approach depicted, you would program things like the cat's ears, whiskers, type of nose, the shape of the face, and colors. You'd describe all these things and have conditions like if the ears are pointy, then a cat; if the ears are sloping down, then possibly a dog, and so on. On the other hand, for a neural network, you code the architecture of the neural network, and then you point the neural network at a folder with all these cats and dogs, with images of cats and dogs which are already categorized. And you tell it to learn. And the neural network will, on its own, understand everything it needs to understand. And then further down, once it's trained up when you give it a new image of a cat or a dog, it'll be able to understand what it was
Here, we have an elementary neural network with one layer. This is called a single-layer feed-forward neural network and it is also called a perceptron.

In Figure 1.1 although the 3 is poorly rendered and sloppily written, our brain can still process that it is a 3. But if we had to write a computer program that takes in a 28 by 28-pixel input of a 3 and have it identify what it is referring to that would be quite a difficult task. That is where the concept of machine learning and neural networks come in.

Structure of a Neural Network

Plain Vanilla form of Neural Networks:
Taking the example of identifying the written integer we will assume that neurons store values between 0 and 1 which represents the grayscale value of each of the pixels mentioned above. Now these 28 x 28 = 784 neurons make up the first layer of the network. The last layer of neurons will consist of 10 integer values that will be the output. There are many hidden layers of neurons in between.

The way the network operates is that activation in one layer determines the activations in consecutive layers and finally, they lead to a result. Activating particular neurons in the first layer causes a particular pattern to propagate in the consecutive layers which then results in a particular pattern in the consecutive layer and so on until there is a pattern seen in the final layer so that our result can be determined.

The function of the middle layers:

They use particular intuitive logical reasoning to pass on information to deduce the required result. The middle layers in our particular example could fire when a group of pixels are lightened up and that may represent a particular line that a few of the integers contain. Each layer acts as a filter in this process and using all the information deduced by each layer a final judgment is made.

Continuing with our example of wanting to determine the integer at hand, if we want a particular edge to be detected at a certain location then we'd have to assign parameters to the network so that the edge pattern is detected and passed on from the first layer to the next.

The parameters that are assigned are weight measures from the connections of neurons from the first layer to the neurons of the second layer. Then we compute the weighted sum of all the activations using each of the assigned weights only for the region that we need to calculate the presence of the edge.

$a_1\mathbf{w_1} + a_2\mathbf{w_2} + a_3\mathbf{w_3} + a_4\mathbf{w_4} + a_5\mathbf{w_5} \ldots.. + a_n\mathbf{w_n}$

Weights can be both positive and negative. To detect the presence of the edge we need the weights in the required region to be positive and the surrounding region to be negative and the calculated sum must be the largest when this condition is satisfied. We also require this weighted sum to be in between the values 0 and 1 so the most meaningful thing to do is pump this sum into a function that squishes its values into the required range [0,1]. A common function that does this is called the sigmoid function:

$$\sigma = \frac{1}{1 + e^{-x}}$$

Also if we only want the particular neuron to light up when it's above a particular value called the "bias" then we subtract that value inside the sigmoid function. If we take the bias to be x:

$\sigma(a_1\mathbf{w_1} + a_2\mathbf{w_2} + a_3\mathbf{w_3} + \ldots.. + a_n\mathbf{w_n} - x)$

The above formula is used to decide which neuron connection will light up. Each of the first layer neurons is connected to each of the neurons of the consecutive layer and each of these connections has a weight and a particular bias. This sum is calculated to determine which particular neuron should light up and when.

This is the process of data propagation through the network of each layer through each neuron. For each different case of problem, we pass into the neural network a different formation of neurons that light up and cause a chain to propagate until they finally converge onto the result using the different weights and biases that are present in each case.

Matrice representation of weight calculation which correlates the first layer to the first neuron(0) of the next layer.

$$\sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix}\right)$$

This can be written in an even more concise manner:

$$a^{(1)} = \sigma\left(\mathbf{W}\mathbf{a}^{(0)} + \mathbf{b}\right)$$

Where W represents the weight matrix and $a^{(0)}$ represents the first neuron of the next layer in the network.

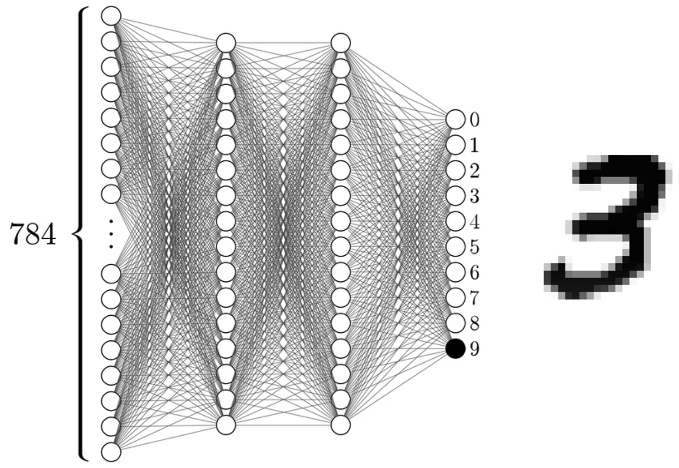


Fig. 1.1

<u>Gradient Descent</u>

The basic concept of gradient descent is finding the minima like a regular calculus problem. We need a method to find out the actual weights and biases that will be used in the given neural network. To start off we basically initialize these weights and biases completely randomly. Then we find a cost function which is our way of calculating the amount of error that we have in our training example. We calculate the cost function as the sum of squares of differences between each of the output activations and the value we require them to have. This is very similar to how to solve a general regression problem.

The larger this value is the more error there is to our network and the less accurate it is. We require a function that will change the values of the initially initialized weights and biases such that the cost functions value is reduced.

The way the network learns is by minimizing the cost function. Gradient descent is a method that helps us descend to the minima of the required function. There are many different forms of gradient descent and we will explore these variants as we move on. Gradient descent is an algorithm that modifies the values of the weights and biases in a way such that the value of the cost function is minimized.

Back Propagation: Neural Network

Learning for a neural network is the process of using the concept of Gradient Descent to calculate the values of weights and biases that will give the most accurate output for any given input. As we know that neural networks are just functions that calculate a particular output based on trained inputs.

Back Propagation is the algorithm for calculating the required gradient for each neuron. The definition of gradient here is the same as it is in 3D geometry. It is a quantity that points in the direction of the greatest increase of a function. Using the negative of the gradient we will be able to reach the minima. If we represent the cost function as a matrix of values then we require the negative of the gradient of the cost function which represents the steps we have to take in reference to every weight and bias used.

$$-\nabla C(\underbrace{\ldots}_{\substack{\text{All weights}\\ \text{and biases}}}) = \begin{bmatrix} 0.20 \\ 0.83 \\ -0.84 \\ \vdots \\ 0.04 \\ 1.57 \\ 1.59 \end{bmatrix}$$

Each value in the given matrix represents how much we need to alter the previous weight and bias values to most effectively decrease the cost function.

Each training example will have its own effect on the values of weights and biases and the average cost function.

Intuitive understanding of the working: If we start out with one particular training example and the effect it has on the cost function after applying gradient descent we will have a basic understanding of how back propagation in neural networks works. We will start out with the final layer and since we want a particular output, the activation of that output should be the highest and the activation of the rest of the neurons should be negligible. This helps us visualize what the gradient descent function needs to do in order for the network to function properly. Using the information for the required output we can then decide what the activations of the previous layers' neurons need to be. This is how neural networks use the concept of back propagation to learn or basically adjust the values of weights and biases to get the required result.

"Neurons that fire together wire together." - This is a statement that we can use to observe the relation between the weights and the activation of each neuron. The higher the activation of the neuron the more it will contribute to the sum and hence the more sensitive the weight will be in deciding the activation of the next neuron. That means that changing the value of this particular weight will have a much higher effect compared to the others.

Since every training example will have its own effect on each weight and bias to activate a particular neuron we have to average out the effects on the cost function gradient so that a more optimal network is made with accurate weights and biases.

In practice, it takes computers a really long time to add up the influence of every single training example in every single gradient descent step. So what's commonly done is: All the training examples are randomly shuffled and grouped into multiple mini-batches. Then we compute a gradient descent step for each mini-batch and update the weights and biases after each mini-batch. This technique is known as 'Stochastic Gradient Descent'.

Back Propagation: Calculus

To understand how we use the chain rule in calculus to depict the concept of back propagation we take the example of a neural network where each layer only has one neuron:

$$C(w_1, b_1, w_2, b_2, w_3, b_3)$$

If we assume the value of activation that we require from the last neuron is y. We get that the cost:

$C_0(\ldots) = (a^{(L)} - y)^2$

$a^{(L)} = \sigma(w^{(L)}a^{(L-1)} + b^{(L)})$

For simplicity we say: $a^{(L)} = \sigma(z^{(L)})$

And $a^{(L-1)}$ is influenced by its own weight and bias and the equations can be traced back to $a^{(0)}$. Now we want to find out how sensitive our cost function is to small changes in weight $w^{(L)}$ and to small changes in $b^{(L)}$. We need to find the values of:

$$\frac{dC_0}{dw^{(L)}} \;\&\; \frac{dC_0}{db^{(L)}}$$

Using the chain rule we can see that

$$\frac{dC_0}{dw^{(L)}} = \frac{dz^{(L)}}{dw^{(L)}} \frac{da^{(L)}}{dz^{(L)}} \frac{dC_0}{da^{(L)}}$$

Substituting the values:

$$\frac{dC_0}{da^{(L)}} = 2(a^{(L)} - y)$$

$$\frac{da^{(L)}}{dz^{(L)}} = \sigma'(z^{(L)})$$

$$\frac{dz^{(L)}}{dw^{(L)}} = a^{(L-1)}$$

We then get the value of $\frac{dC_0}{dw^{(L)}}$ to be:

$$\frac{dC_0}{dw^{(L)}} = a^{(L-1)} \sigma'(z^{(L)})\, 2(a^{(L)} - y)$$

We can see that the activation value of the previous neuron plays a large role in the sensitivity of cost function to the change in weight. This is where the idea of "neurons that fire together wire together comes in". The value calculated above is only the change in the cost for a particular training example. To find the effect of all the weights of training examples we need to sum up all the derivatives and take their average:

$$\frac{dC}{dw^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{dC_k}{dw^{(L)}}$$

And this will only be one term in the gradient of the cost function which in itself will be a ss.

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w^{(1)}} \\ \frac{\partial C}{\partial b^{(1)}} \\ \vdots \\ \frac{\partial C}{\partial w^{(L)}} \\ \frac{\partial C}{\partial b^{(L)}} \end{bmatrix}$$

Now calculating the sensitivity of the cost function to the bias:

$$\frac{dC_0}{db^{(L)}} = \frac{dz^{(L)}}{db^{(L)}} \frac{da^{(L)}}{dz^{(L)}} \frac{dC_0}{da^{(L)}}$$

Two of the terms in the chain rule are still the same. Calculating the third term:

$$\frac{dz^{(L)}}{db^{(L)}} = 1$$

We can see that the variation of the cost function with respect to the bias is:

$$\frac{dC_0}{dw^{(L)}} = 1 \times \sigma'(z^{(L)})\, 2(a^{(L)} - y)$$

We can also calculate the variation of the cost function with respect to the previous activation which will come out to be:

$$\frac{dC_0}{da^{(L-1)}} = w^{(L)}\, \sigma'(z^{(L)})\, 2(a^{(L)} - y)$$

From here we can use the concept of back propagation and visualize how the cost function of each training example is affected by all the previous activations. We can see how sensitive the cost function is to the previous weights and the previous biases.

Expanding upon the example:

Although we took an elementary example in which each layer contained only one neuron, one might assume that the problem gets exponentially more difficult when there are a wide number of neurons involved, but that is not the case. Instead of a neuron's activation being $a^{(L)}$, it will also contain a subscript that denotes which neuron of the layer it is in The cost function will be the sum of squares of the differences in the value of the neuron from the cost function for each neuron in the layer. While calculating the derivative of the cost function to the weights we will have to take the sum of the chain rule derivatives since the change in the weight now affects more than one neuron.

The modified equations in the gradient will be:

$$\nabla C \longleftarrow \begin{cases} \dfrac{\partial C}{\partial w_{jk}^{(l)}} = a_k^{(l-1)}\sigma'(z_j^{(l)}) \boxed{\dfrac{\partial C}{\partial a_j^{(l)}}} \\ \boxed{\begin{array}{c} \displaystyle\sum_{j=0}^{n_{l+1}-1} w_{jk}^{(l+1)}\sigma'(z_j^{(l+1)})\dfrac{\partial C}{\partial a_j^{(l+1)}} \\ \text{or} \\ 2(a_j^{(L)} - y_j) \end{array}} \end{cases}$$

Artificial neural network

ANN is a neural network derived from the functioning of the human brain. It is always configured for specific applications like pattern detection or data classification. They consist of layers of interconnected "neurons" that process and transmit information.

ANNs have densely interconnected small units known as neurons and each unit takes a real-valued input and gives out a real-valued output.

There are several different frameworks for ANNs including Feedforward neural networks, Recurrent neural networks, Convolutional neural networks, and Autoencoders. In this paper, we will be discussing RNN in particular.

Interconnections:

These are defined as the way the neurons in ANN are connected to each other. These arrangements always have two layers, namely the input layer and the output layer. There are also layers of neurons that are neither part of the input or output layer. These layers are known as hidden layers which were previously discussed.

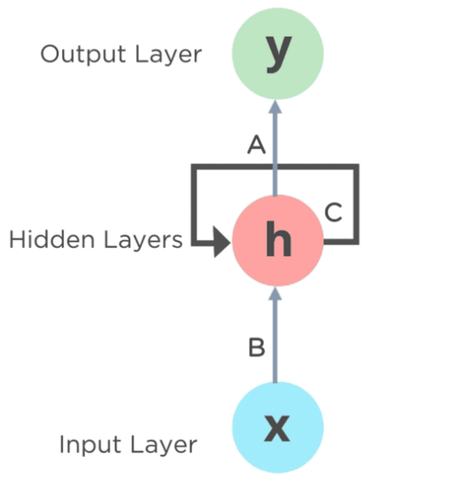
The neural network structure initially discussed is the most simple form of a neural network whose architecture is referred to as a Feedforward neural network where the information flows through the neurons during computation only in the forward direction.

Recurrent neural network

Recurrent neural networks are a form of artificial neural networks in which the flow of information of a neuron is not only in a single direction. It is a network that takes the output from the previous step and uses it as input for the next step. It also retains information from the past and uses that information to process new input.

Recurrent Neural Networks (RNNs) are artificial neural networks that work with time series or sequential data.

They are commonly used for ordinal or temporal problems like language translation, natural language processing(NLP), speech recognition, and image captioning. Generally in standard NLP techniques like BoW, Word2Vec, or TF-IDF, we lose the structure of the sentence. This is where the concept of RNN comes in.



A, B and C are the parameters

RNNs are well-suited for tasks that involve sequential data. The basic structure of any neural network includes an input layer, one or multiple hidden layer(s), and an output layer. The above picture depicts the hidden layers compressed into one circle. The following picture depicts the decompressed version of information flow through the hidden layers:

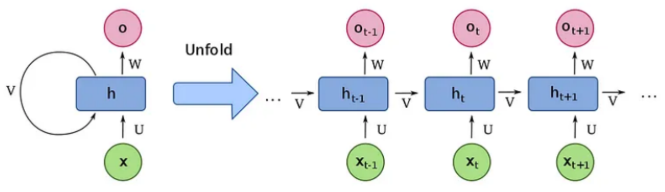
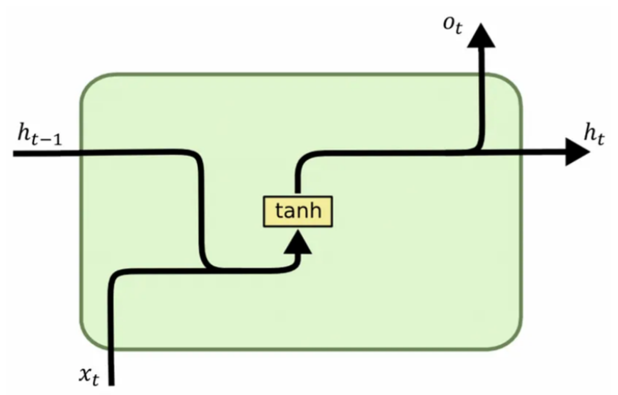


Fig 2: Unfolded RNN layer

The above image shows how the output of the previous layer is fed into the next layer for the purpose of making predictions and repeating the same structure.

In the above image
X: Input, usually a piece of text
O: Output, also represents a piece of text generally
h(t): Represents a hidden state at time t and acts as the “memory” of the network.
V: Communication from one-time step to the other.
A simple RNN neuron is represented as follows:



The flow of information through this block is represented by the arrows. The new input along with information from the previous neuron is taken as input for the block and after passing it through an activation function such as tanh an output is given out along with the input for the next block represented by O_t and h_t.

In the above block, the output is calculated based on the following equations:

$$a_t = X_t U + h_{t-1} V + b$$
$$h(t) = f(a_t)$$
$$O_t = c + W h_t$$
$$\hat{y}_t = softmax(O_t)$$

Here f represents the activation function. Eg: tanh. h(t) is calculated based on the current input X(t) and the previous time step's hidden state h(t-1). U is the input weight vector representing the weights of all the connections and b is the bias vector representing the biases for each step.

The softmax function, also known as softargmax or normalized exponential function, converts a vector of K real numbers into a probability distribution consisting of K probabilities proportional to the exponentials of the input numbers. This is explained later in the paper.

The output O(t) is calculated in a similar fashion to h(t) where W is the weight vector for calculating the output function based on the calculated hidden state value for that particular time and c is the corresponding bias vector. y(t) is a function that represents the probability distribution of all the values of the output function O(t).

X(t): The current input at time t
h(t-1): Value of previous time step's hidden state
a(t): Weighted sum of the inputs with the bias which helps calculate h(t)
U: Weight vector representing the weights for the corresponding input X(t)
V: Weight vector representing the weights for the previous hidden state's values h(t-1).
f: Activation function such as tanh, relU, or sigmoid
O(t): Output value of time state t
c: Bias Vector for output calculation
W: Weight vector for output state calculation
b: Bias vector for hidden state value calculation

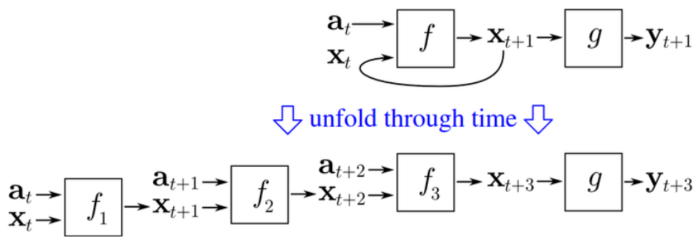
Backpropagation through time (BPTT)

This is the learning step for RNN which is very similar to the backpropagation we have seen in a Feedforward neural network. We use backpropagation to set the weights and biases using the value of the gradient. We know that the value of the gradient at time t is dependent on the value of the gradient at all the times after time t: t+1, t+2, etc. So this is where the concept of BPTT comes in.

If we assume the training data for our RNN is an ordered sequence of k input-output pairs: (a_0,y_0), (a_1,y_1) …. We specify an initial value for the hidden state x_0.

BPTT begins by unfolding a recurrent neural network in time. The unfolded network contains k inputs and outputs, but every copy of the network shares the same parameters. Then the backpropagation algorithm is used to find the gradient of the cost with respect to all the network parameters.

Consider an example of a neural network that contains a recurrent layer f and a feedforward layer g. There are different ways to define the training cost, but the aggregated cost is always the average of the costs of each of the time steps. The cost of each time step can be computed separately. The figure below shows how the cost at time t+3 can be computed, by unfolding the recurrent layer f for three time steps and adding the
feedforward layer g. Each instance of f in the unfolded network shares the same parameters. Thus the weight updates in each instance (f_1, f_2, f_3) are summed together.



Sentiment Analysis using Simple RNN

To start at the beginning let's take a look at the Google dictionary definition of sentiment analysis: **Sentiment analysis** is the use of natural language processing, text analysis, computational linguistics, and biometrics to systematically identify, extract, quantify, and study affective states and subjective information. Due to the internal memory feature of RNNs, which recalls both previous sequences and current input, they are able to capture context rather than simple individual words. It can be used to analyze social media posts, customer reviews, or any other text data to classify it as positive, negative, or neutral.

One approach to sentiment analysis is to use a SimpleRNN (Simple Recurrent Neural Network) layer, which is a type of recurrent neural network (RNN). RNNs are well-suited for sequence data, as they can capture dependencies between elements in a sequence.

Data Preparation:

- Collect a labeled dataset of tweets with sentiment labels (positive, negative, or neutral). Let's assume we

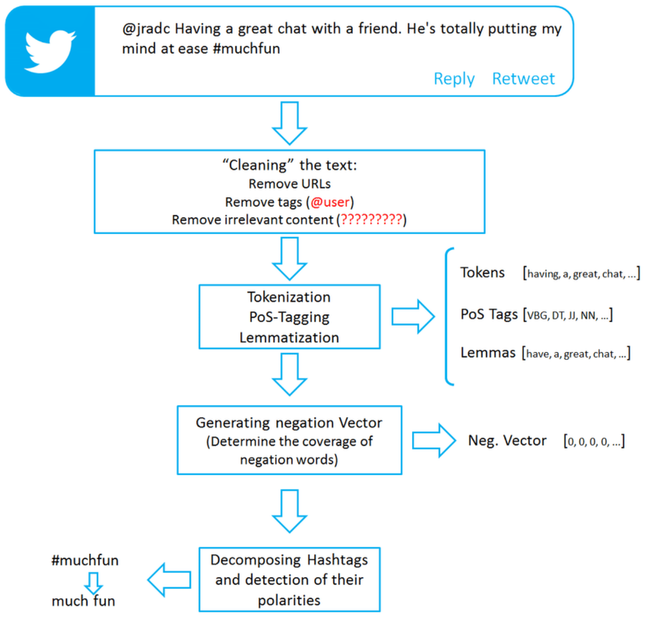
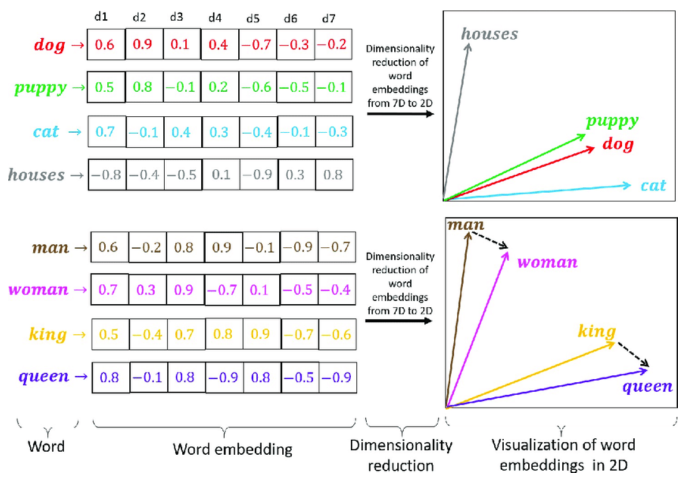

@jradc Having a great chat with a friend. He's totally putting my mind at ease #muchfun
Reply
Retweet
"Cleaning" the text:
Remove URLs
Remove tags (@user)
Remove irrelevant content (?????????)
Tokenization
PoS-Tagging
Lemmatization
Tokens [having, a, great, chat, ...]
PoS Tags [VBG, DT, JJ, NN, ...]
Lemmas [have, a, great, chat, ...]
Generating negation Vector (Determine the coverage of negation words)
Neg. Vector [0, 0, 0, 0, ...]
Decomposing Hashtags and detection of their polarities
#muchfun
much fun

figure 1.2

- Have a dataset with 1,000 labeled tweets.
- Preprocess the tweets by removing hashtags, mentions, URLs, and special characters.
- Tokenize the preprocessed tweets to split them into individual words or tokens.

Word Embedding:

- Use a pre-trained words embedding model like Word2Vec or GloVe to represent each word in the tweets as dense vectors. Let's assume we are using GloVe embeddings.
- Map each word in the vocabulary to its respective GloVe embedding vector. For example, the word "good" might be mapped to [0.4, 0.2, -0.1, ...] in a 300-dimensional vector space.



	d1	d2	d3	d4	d5	d6	d7
dog →	0.6	0.9	0.1	0.4	−0.7	−0.3	−0.2
puppy →	0.5	0.8	−0.1	0.2	−0.6	−0.5	−0.1
cat →	0.7	−0.1	0.4	0.3	−0.4	−0.1	−0.3
houses →	−0.8	−0.4	−0.5	0.1	−0.9	0.3	0.8
man →	0.6	−0.2	0.8	0.9	−0.1	−0.9	−0.7
woman →	0.7	0.3	0.9	−0.7	0.1	−0.5	−0.4
king →	0.5	−0.4	0.7	0.8	0.9	−0.7	−0.6
queen →	0.8	−0.1	0.8	−0.9	0.8	−0.5	−0.9

Dimensionality reduction of word embeddings from 7D to 2D
houses
puppy
dog
cat
man
woman
king
queen
Word
Word embedding
Dimensionality reduction
Visualization of word embeddings in 2D

Padding and Sequences:

- Since tweets can have varying lengths, pad or truncate the sequences to a fixed length. Let's set the maximum sequence length to 20.
- For a tweet with fewer than 20 words, pad it with a special padding token. For a tweet longer than 20 words, truncate it to the first 20 words.

Building the Sentiment Analysis Model:

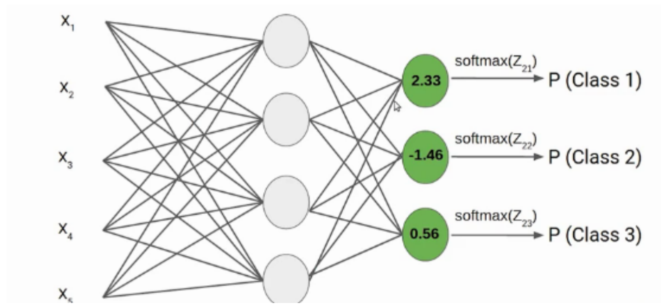
- Initialize a SimpleRNN layer in Keras with, say, 100 hidden units. The input shape would be (20, 300) to match the padded sequence length and the dimension of GloVe embeddings.

- Connect the SimpleRNN layer to a fully connected (dense) output layer with three units for the three sentiment classes (positive, negative, neutral).
- Apply the softmax activation function on the output layer to obtain class probabilities. Without an activation function, a neural network is a simple linear regression model. This means the activation function gives non-linearity to the neural network.

  Let's see how the softmax activation function actually works. Similar to the sigmoid activation function, the SoftMax function returns the probability of each class. Here is the equation for the SoftMax activation function.

$$softmax(z_i) = \frac{exp(z_i)}{\sum_j exp(z_j)}$$

Here, the Z represents the values from the neurons of the output layer. The exponential acts as the nonlinear function. Later these values are divided by the sum of exponential values in order to normalize and then convert them into probabilities.



Training the Model:

- Split the dataset into training and testing sets, e.g., 80% for training and 20% for testing.
- Compile the model using the categorical cross-entropy loss function and an optimizer like Adam.
  - Train the model using the training set, adjusting the model's parameters (weights and biases) through backpropagation and gradient descent
- Monitor the training process and evaluate the model's performance on the testing set.

Prediction:

- Preprocess a new, unseen tweet by removing hashtags, mentions, URLs, and special characters, and tokenize it.
- Map the tokens to their respective GloVe embedding vectors.
- Pad or truncate the sequence to match the fixed length of 20.
- Pass the preprocessed and padded tweet through the trained model.
- The model will output a probability distribution over the sentiment classes (e.g., [0.2, 0.7, 0.1]), indicating the predicted sentiment probabilities.

Evaluation:

- Evaluate the model's performance on the testing set using metrics like accuracy, precision, recall, or F1-score.
- Adjust the model architecture, hyperparameters, or preprocessing steps if necessary to improve the sentiment analysis performance.

In practice, you may need to handle additional challenges like handling imbalanced classes, experimenting with different model architectures, or incorporating additional features to improve the model's performance.
Additionally, techniques like regularization and dropout can be employed to prevent overfitting.