Heriot-Watt University

Doctoral Thesis

Multimodal Representation Learning

Author:
Eli Sheppard

Supervisors:
Dr. Katrin Lohan
Dr. Oliver Lemon

Doctoral Thesis submitted in fulfilment of the requirements for the degree of PhD Robotics and Autonomous Systems

in the

Edinburgh Centre for Robotics

July 2019

Declaration of Authorship

I, Eli Sheppard, declare that this theis titled, 'Multimodal Representation Learning' and the work presented in it is my own. I confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: ____

Date: ____

Abstract

Acknowledgements

I would like to thank Katrin and Oliver for their continued support throughout my academic career. I would also like to thank Ingo Keller for his invaluable lessons on the art of Python programming.

Contents

Abbreviations

CNN	**C**onvolutional **N**eural **N**etwork
AE	**A**uto**e**ncoder
MAE	**M**ultimodal **A**uto**e**ncoder
RNN	**R**ecurrent **N**eural **N**etwork
GRU	**G**ated **R**ecurrent **U**nit
LSTM	**L**ong **S**hort **T**erm **M**emory
YARP	**Y**et **A**nother **R**obotics **P**latform

Chapter 1

A Primer on Artifical Neural Networks

Introduction

This chapter presents an overview of the mathematical theory behind artificial neural networks and how they learn. It should serve as a quick guide to the techniques used in the experiments within this thesis.

Perceptrons

The perceptron is the parent of modern artificial neurons **?**. Perceptrons arranged in layers, referred to as multi-layer perceptrons, are therfore the predecessor of modern artificial neural networks.

What is a Perceptron

Perceptrons are biologically inspired computation units and are a type of artificial neuron. They take a series of binary inputs, compute a weighted sum and produce a binary output based on the value of this sum as seen in figure 1.1.

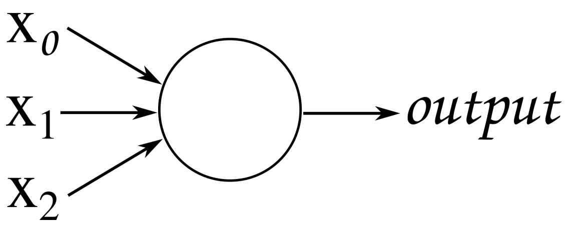

FIGURE 1.1: A perceptron with three binary inputs, x_0, x_1, x_2.

$$output = \begin{cases} 0 & \text{if } \sum_j w_j x_j + b < 0 \\ 1 & \text{if } \sum_j w_j x_j + b \geq 0 \end{cases} \tag{1.1}$$

Where x_j is the jth input, w_j is it associated weight and b is a constant value called the bias, which affects how easy it is for the neuron to activate. The output of the perceptron is caluclated using the formulation shown in 1.1.

By adjusting each of the weights we can change the output of the perceptron. For example, if we wanted to use a perceptron to decide whether to have a picnic today we can select a set of relevant inputs, “the weather is nice” and “the pollen count is low”.

If it is a sunny day, and the pollen count is high, the input to the perceptron would be $[1, 0]$.

We will set our weights depending on how important each of the inputs is. No one likes a picnic in the rain, so the weather is important, whilst the pollen count is only important if you suffer from hayfever. We will select a bias of -5 for our perceptron.

For person A, the weights might look like $[7, 0]$ (person A doesn’t suffer from hayfever). Therefore the output of the perceptron would be 1 as $1 \times 7 + 0 \times 0 - 5 = 2$ is greater than 0, so person A will go for a picnic.

Person B owns a large umbrella (so the weather doesn’t matter as much) but they do suffer from hayfever, their weights might look like $[4, 7]$. The output of the perceptron would be 0 as $1 \times 4 + 0 \times 7 - 5 = -1$ is less than 0. Therefore, person B would wait for a day with a lower pollen count for a picnic.

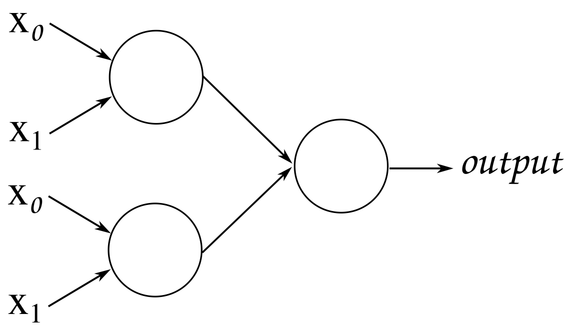

FIGURE 1.2: A multi-layer perceptron consisting of three perceptrons arranged in two layers, with three binary inputs, x_0, x_1, x_2.

Multi-Layer Perceptron

In the previous section I demonstrated how a single perceptron can be used to make decisions based on a set of inputs. However, things get much more interesting when we start to link multiple perceptrons together into multi-layer perceptrons (MLP).

If we take the example from the previous section about deciding to go for a picnic or not, we can use the MLP shown in 1.2 to consider what happens if person A and B want to go for a picnic together.

Given the previous conditions of a sunny day with a high pollen count, person A wants to go whilst person B does not, as demonstrated by the outputs of their individual perceptrons. Taking these outputs as inputs to the second layer of our MLP, we can see whether the picnic will go ahead.

This time we will set the weights of the second layer based on who shouts the loudest. In this case, person A really wants to go and is very vocal about it. $1 \times 9 + 0 \times 5 - 5 = 4$ so the picnic will go ahead. This resulted in person B developing a headache and sneezing a lot. Clearly we need to find a way to adjust the weights of our MLP so that it makes better decisions in the future.

Activation Functions

Only being able to handle binary values is a major drawback of the perceptron. An artificial neuron which can handle continuous values is much more useful.

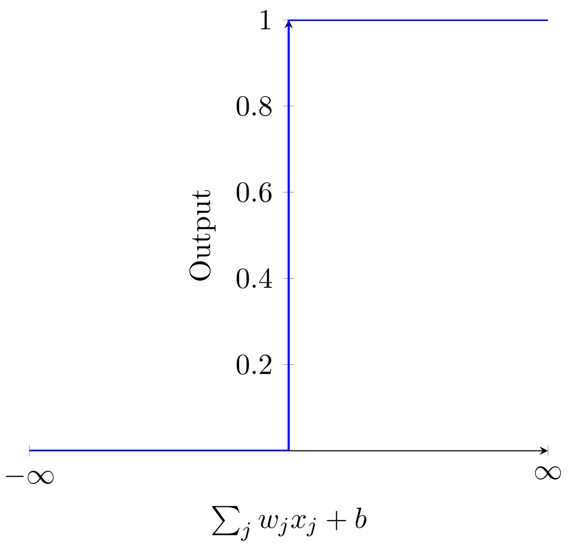

FIGURE 1.3: Visualisation of the perceptron activation function.

This limitation occurs due to the activation function of the perceptron shown in equation 1.1. Visualising the perceptron activation function highlights that changes in the output of the neuron are not proportional to changes in the weights of the neuron as seen in figure 1.3. With a few small changes we can make an artifical neuron that accepts continuous values and has a continuous output.

A continuous output is very important as it means that a small change in the weights of a neuron will cause a small change in its output. This makes it much easier to understand how changing the weights affects the output of the network as the change in output becomes proportional to the change in weights as shown in equation 1.2

$$\delta Output \propto \delta W \tag{1.2}$$

Sigmoid Neurons

The sigmoid neuron uses the sigmoid function, shown in equation 1.3 as its activation function.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{1.3}$$

z is the sum of the weights multiplied by their respective inputs as seen in equation 1.4.

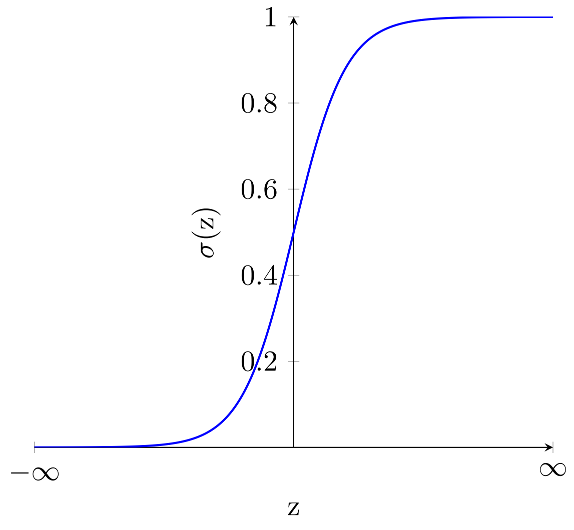

FIGURE 1.4: Visualisation of the sigmoid activation function.

$$z = \sum_j w_j x_j + b \tag{1.4}$$

As we can see in figure 1.4, as z changes, their is a proportional change in the output $\sigma(z)$, unlike in figure 1.3 where the output only changes when z crosses the y-axis.

Now that we have an activation function that can produce continuous values, we can consider a much more diverse range of data to train our neural networks with. So instead of just making yes or no decisions we can look at, for example, the colours of pixels and decide if there is a cat in the image or given todays weather, predict if it will rain tomorrow.

Other activation functions

There are two other important and commonly used activation functions, though many others exist. These are, the hyperbolic tangent (Tanh) shown in figure 1.5 and rectified linear unit (Relu) shown in figure 1.6.

The Tanh function looks similar to the sigmoid function, however it has an output between negative one and one, where the sigmoid goes from zero to one. This is useful when having negative values within the network is important.

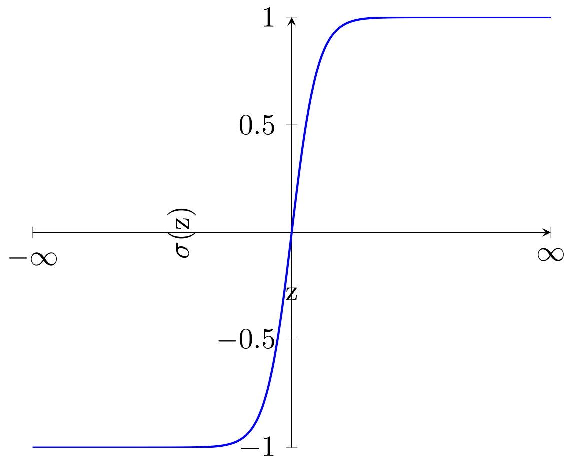

FIGURE 1.5: Visualisation of the hyperbolic tangent activation function.

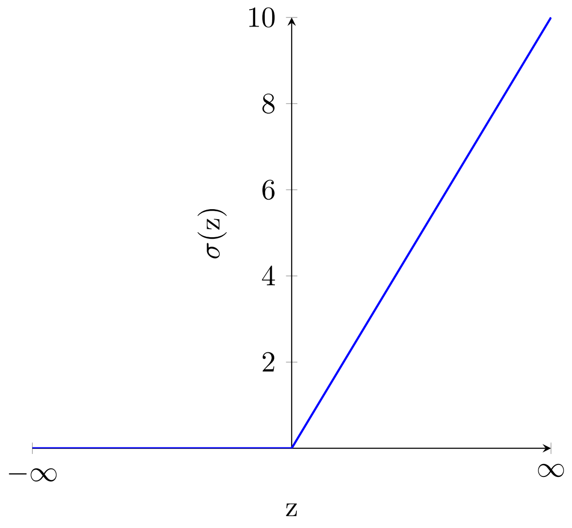

FIGURE 1.6: Visualisation of the rectified linear unit activation function.

The Relu was created to address and issue known as vanishing gradients. This is to do with how neural networks are trained, as expalined in the next section. Briefly, as training depends on error gradients, having a linear activation function means that deeper networks [1] can be trained as non-linear functions like the sigmoid or Tanh will have reduced gradient magnitude when they are differentiated to backpropogate the error through the network. If that doesn't make sense yet, don't worry, it will become clear shortly.

[1]Deeper networks are beneficial as they can make more complex decisions by layering more and more simple decsions as shown in the MLP example in section1.2

Learning Algorithms

In section 1.2 we saw how simple computation units can make simple decisions and how these can be chained together to make more complex decisions. However, sometimes the decisions we make are wrong and we need to learn from our mistakes.

In general, perceptrons (and their more modern decendents) will have their initial weights set randomly, unlike in our example where I chose them.

Randomly setting weights is useful in practice as we will usually have large numbers of inputs, weights, layers and artifical neurons, so setting each by hand would be intractable. Also, if we knew what weights to set, we wouldn't need a neural network or a learning algorithm to solve our problem as we would likely have a more efficient mathematical description of the problem [2]

With random weights our networks will make a lot of wrong decisions! However if we have a smart way of adjusting our weights we can make our neural networks get better at making decisions over time. This improvement is what I am referring to when I say "machine learning".

Types of Training

There are many ways to train neural networks, supervised, unsupervised, reinforcement or adversarial learning (to name a few). The experiments in this thesis will focus on supervised and unsupervised learning.

Supervised learning - an external system is used to provide ground truth values for the desired output of the network. An example would be classification using standardised datasets like MNIST handwritten digits LeCun [1998].

Unsupervised learning - no external system provides ground truth values, instead these are inferred directly from the data. Autoencoders (discussed in great deal in later chapters) are a good example of this.

[2]there is a large body of work concerning the best way to initialise neural networks, I will briefly comment on this in section **??**.

Cost functions: How wrong am I?

In order for our neural networks to learn, they need feedback as to how wrong they are. This is done using a cost function, a formula which gives a measure of how good or bad our network is at a particular task.

Depending on the particular method we use to train our networks, the cost function may be given a different name. For example in reinforcement learning, the cost function is typically called a value function. However, they all serve the same purpose, which is to provide feedback as to how well our network is doing at the task it has been assigned. Therefore, to avoid jargon I will refer to this as a cost or loss function throughout this thesis.

The cost fuction can take on many forms, in supervised and unsupervised training alike. Here are a few of the more commonly used ones and an explanation of each.

Mean Squared Error

Mean squared error (MSE) measures the average difference between points as shown in equation 1.5.

$$C = \frac{1}{n}\sum_{i=0}^{n}(Y_i - \hat{Y}_i)^2 \tag{1.5}$$

When used as a loss function, MSE gives a measure of how bad our estimate $\hat{Y}_i$ is of our true value Y_i on average for for all n training samples in a given dataset. A perfect model would have an MSE of 0 as $\hat{Y}_i$ would be equal to Y_i for all values of i.

In general, Y_i and $\hat{Y}_i$ can be of arbitrary size and shape (as long as they match each other). That is, they can be scalers, vectors or matricies.

Cross-Entropy

Cross-entropy measures the error between two probability distributions P and Q using the formula shown in equation **??**. P is the true probability distribution which we are trying to model with our predicted probability distribution Q.

$$C = -\sum_{i=0}^{n} P(i)log(Q(i)) \tag{1.6}$$

When using cross-entropy as a loss function, we can view the output of our neural network as a probability distribution. For example, in a multiclass classification problem with k classes, each of the k outputs of the network tells us the probability that the network believes an input to belong to class k.

Here is a more concrete example: Given a dataset containing images of animals: cats, dogs, horses and snakes. A single image containing a cat would be labelled $[1, 0, 0, 0]$. This tells us the exact probability distribution of the image, P i.e. the image certainly contains a cat and does not contain any other animals.

now as our network trains, it may give a prediction of $[0.5, 0.3, 0.15, 0.05]$, this is our predicted probability distribution Q.

Calculating our cross-entropy we get:

$$C = -(1log(0.5) + 0log(0.3) + 0log(0.15) + 0log(0.05) = 0.301_{3s.f.}) \tag{1.7}$$

If we then do some training using gradient descent our network might then predict $[0.8, 0.2, 0, 0]$ giving a cross-entropy of $0.0969_{3s.f.}$. Clearly this represents an improvement, as the cost is closer to zero and our prediction is better. The network believes with 80% certainty that the images is of a cat.

Kullback-Leibler Divergence

Another method for defining the difference between two distributions is the Kullback-Leibler Divergence (KLD).

KLD is a non-symetric distance measure between two distributions, P and Q. As it is non-symetric $D_{KL}(P||Q) \neq D_{KL}(Q||P)$ unless $P = Q$.

$$D_{KL} = -\int_{i=0}^{n} P(i)Log(\frac{Q(i)}{P(i)}) \tag{1.8}$$

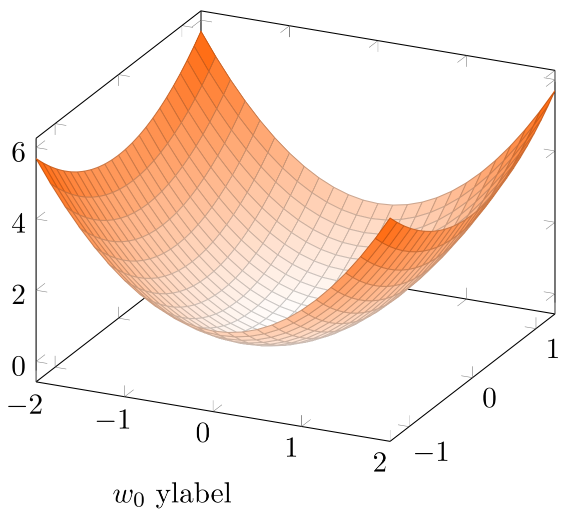

FIGURE 1.7: The cost landscape of a bivariate function

KLD is useful in machine learning when we wish to calculate how different a predicted distribution is from a desired distribution. Most commonly, we will want P to be a Gaussian distribution, though which distribution is chosen will depend on the specific problem. We will then use the KLD as a constraint to force our networks to make predictions which follow as closely as possible to this desired distribution, i.e. we will minimse $D_{KL}(P||Q)$.

Typically KLD is not used as a cost function on its own, but is used to constrain the output of a single layer of a network (usually not the output) whilst also minimising another cost function.

The best example of this is a Variational AutoEncoder (VAE), which will be discussed later.

Gradient Decent

Now that we have a method for determining how well our neural network performs at a task, we can observe how this changes as the weights of the network are changed.

To make things simple, lets assume our network only has two weights, though the following derivation generalises to any number of weights. This could give us a cost landscape like that shown in figure 1.7. Our aim is to minimise the cost as the cost is a measure of how wrong our network is.

In more definate terms we wish to minimise equation 1.9, where Y_i is the desired output value for input i and z is as defined in equation 1.4.

$$C(w_0, w_1) = \frac{1}{n}\sum_{i=0}^{n}(Y_i - \sigma(z_i))^2 \tag{1.9}$$

This is the mean squared error rewritten slightly to highlight the dependence on the weights of the network.

If we make a small change to either of the weights w_0 and w_1 we will get a small change in the cost, C as seen in equation 1.10.

$$\Delta C \approx \frac{\delta C}{\delta w_0}\Delta w_0 + \frac{\delta C}{\delta w_1}\Delta w_1 \tag{1.10}$$

We want to minimise C so we want ΔC to be negative. To make ΔC negative we will need to first denote that the gradient of C, ∇C.

$$\nabla C \equiv (\frac{\delta C}{\delta w_0}, \frac{\delta C}{\delta w_1})^T \tag{1.11}$$

In equation 1.11 we can see that the gradient of C is equivalent to the partial derivatives of C with repect to the weights of the network. For simplicity we will collect the changes in weights into a single vector $\Delta w \equiv (\Delta w_0, \Delta w_1)^T$.

By substituing this into equation 1.10 we get equation 1.12.

$$\Delta C \approx \nabla C \cdot \Delta w \tag{1.12}$$

Looking at equation 1.12, we can see that to make ΔC negative we can set Δw as in equation 1.13, where η is a small positive constant called the learning rate.

$$\Delta w = -\eta \nabla C \tag{1.13}$$

By substituting equation 1.13 into equation 1.12 we can see that $\Delta C = -\eta|\nabla C|^2$ and because $|\nabla C|^2 \geq 0$ this gaurantees that C will always decrease if the weights are changed in accordance to equation 1.13.

By iteratively making changes to the weights according to 1.13 we will eventually reach the minima of C so long as we select an appropriate learning rate [3].

Backpropagation

With a gradient decent providing a method for adjusting weights, we now only need one more ingredient to be able to train our neural networks. Whilst our cost function tells us about the error of our network at its output, we need a method to calculate the error for any neuron in any layer - this is exactly what backpropagation does.

Backpropogation is defined by four equations: 1)An equation for the error in the output layer, δ_L:

$$\delta_L = \nabla_a C \circ \sigma'(z_L) \tag{1.14}$$

Equation 1.14 shows the output error δ_L equates to changes in the cost C with respect to its activations a and the derivative of the output layers activation function σ' when the input to that layer is z_L.

2)An equation for the error δ_l in terms of the error in the next layer:

$$\delta_l = ((w_{(l+1)})^T \delta_{(l+1)} \circ \sigma'(z_l) \tag{1.15}$$

where $(w_{(l+1)})^T$ is the transpose of the weight matrix for the layer above. Multiplying the error from the layer above by the transposed weight matrix can be seen as moving the error backward through that weight matix and the Hadamard product[4] $\circ$ with $\sigma'(z_l)$ moves the error backward through the activation function.

[3]Learning rate is discussed in more detail in section 1.4.5.1

[4]The Hadamard product is the elementwise product of two matricies as shown in equation 1.16.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \circ \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae & bf \\ cg & dh \end{bmatrix} \tag{1.16}$$

3)An equation for the rate of change of the cost with respect to any bias in the network:

$$\frac{\delta C}{\delta b_l j} = \delta_l j \tag{1.17}$$

where $\frac{\delta C}{\delta b_l j}$ is the rate of change of the cost C with respect to bias b of neuron j, in layer l and $\delta_l j$ is the j^{th} entry in the error matrix δ for layer l i.e. the error for the $j^t h$ neuron in layer l.

4)An equation for the rate of change of the cost with respect to any weight in the network:

$$\frac{\delta C}{\delta w_{ljk}} = a_{(l-1)k} \delta_l j \tag{1.18}$$

where $\frac{\delta C}{\delta w_{ljk}}$ is the rate of change of cost C with respect to the change in weight w_k of neuron j in layer l and $a_{(l-1)k}$ is its input activation.

Using these four equations, 1.14, 1.15, 1.17 and 1.18, we can calculate the change in error due to any weight or bias in the network and thus can use gradient decent to optimise its value to reduce the error at the output of the network.

One consequence of the backpropagation algorithm is that we are limited in terms of the trainable depth of our networks. As we continually differentiate the error to pass it back through each layer and multiply it by transposed weight matricies, the magnitude of the error can shrink. This results in vanishing error gradients the deeper into the network we go. This mean that at some point, the error with respect to a weight or bias will appear to be zero (or approxiamtely zero) and thus we cannot adjust it by gradient decent, so we are limited to not having infinitely deep networks.

Extensions and Improvements

There are many extensions to simple gradient decent based learning, here are a few to be aware of.

Learning Rate Schedule

Depending on how far we are from the global error minima, we may wish to change the value of the learning rate η. Recall from equation 1.13 $\Delta w = -\eta \nabla C$, that the learning rate determines how big of a step we take, following the gradient of the error, each time we update the weights of our neural network.

If we are far from the minimum point for the cost with respect to the weights, we may wish to take larger steps, so that we can more quickly reduce the total cost. However, as we get closer we want to ensure that we do not step over the minimum.

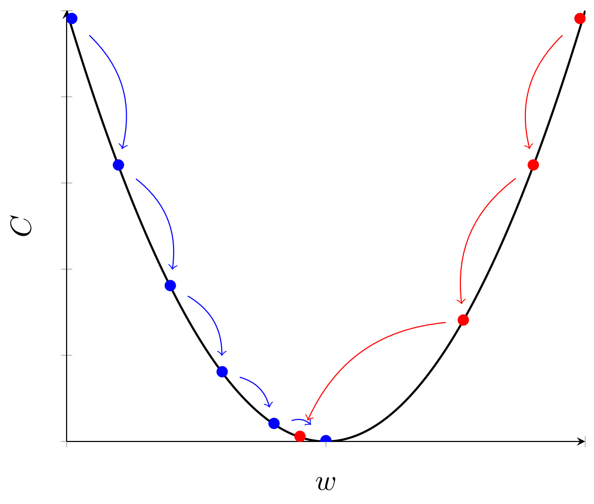

Figure 1.8: Visualisation of cost minimisation in two dimensions for different learning rate schedules. Blue: decreaseing learning rate, Red: large fixed learning rate.

Momentum

Convolutional Neural Networks

What is Convolution?

Kernels

Strides

Dilations

Transposed Convolutions

Recurrent Neural Networks

Vanilla RNN

Gated RNN

Summary

Bibliography

Barsalou, L. W. (2008). Grounded cognition. *Annu. Rev. Psychol.*, 59:617–645.

Hammami, N. and Bedda, M. (2010). Improved tree model for arabic speech recognition. In *2010 3rd International Conference on Computer Science and Information Technology*, volume 5, pages 521–526. IEEE.

Hammami, N. and Sellam, M. (2009). Tree distribution classifier for automatic spoken arabic digit recognition. In *2009 International Conference for Internet Technology and Secured Transactions,(ICITST)*, pages 1–4. IEEE.

LeCun, Y. (1998). The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*.

Ma, W. J., Zhou, X., Ross, L. A., Foxe, J. J., and Parra, L. C. (2009). Lip-reading aids word recognition most in moderate noise: a bayesian explanation using high-dimensional feature space. *PLoS One*, 4(3):e4638.

McGurk, H. and MacDonald, J. (1976). Hearing lips and seeing voices. *Nature*, 264(5588):746.

Ngiam, J., Khosla, A., Kim, M., Nam, J., Lee, H., and Ng, A. Y. (2011). Multimodal deep learning. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 689–696.

Samuel, A. G. (1997). Lexical activation produces potent phonemic percepts. *Cognitive psychology*, 32(2):97–127.