[MITx 6.86x Notes Index]

Unit 03 - Neural networks

Lecture 8. Introduction to Feedforward Neural Networks

8.1. Unit 3 Overview

This unit deals with Neural Networks, powerful tools that have many different usages as self-driving cars or playing chess.

Neural networks are really composed of very simple units that are akin to linear classification or regression methods.

We will consider:

- feed-forward neural networks (simpler)
- recurrent neural networks: can model sequences and map sequences to class labels and even to other sequences (e.g. in machine translation)
- convolutional neural network: a specific architecture designed for processing images (used in the second part of project 2 for image classification)

At the end of this **unit**, you will be able to

- Implement a feedforward neural networks from scratch to perform image classification task.
- Write down the gradient of the loss function with respect to the weight parameters using back-propagation algorithm and use SGD to train neural networks.
- Understand that Recurrent Neural Networks (RNNs) and long short-term memory (LSTM) can be applied in modeling and generating sequences.
- Implement a Convolutional neural networks (CNNs) with machine learning packages.

8.2. Objectives

At the end of this **lecture**, you will be able to

- Recognize different layers in a feedforward neural network and the number of units in each layer.
- Write down common activation functions such as the hyperbolic tangent function `tanh`, and the rectified linear function `ReLU`.

- Compute the output of a simple neural network possibly with hidden layers given the weights and activation functions .
- Determine whether data after transformation by some layers is linearly separable, draw decision boundaries given by the weight vectors and use them to help understand the behavior of the network.

8.3. Motivation

The topic of feedforward neural networks is split into two parts:

- part one (this lesson): the model
    - motivations (what they bring beyond non-linear classification methods we have already seen);
    - what they are, and what they can capture;
    - the power of hidden layers.
- part two (next lesson): learning from data: how to use simple stochastic gradient descent algorithms as a successful way of actually learning these complicated models from data.

Neural networks vs the non-linear classification methods that we saw already

Let's consider a linear classifier $\hat{y} = sign(\theta \cdot \phi(\mathbf{x}))$.

We can interpret the various dimensions of the original feature vector $\mathbf{x} \in \mathbb{R}^d$ as d nodes. Then these nodes are mapped (non necessairly in a linear way) to D nodes of the feature representation of $\phi(\mathbf{x} \in \mathbb{R}^D)$. Finally we take a linear combination of these nodes (dimensions), we apply our sign function and we obtain the classification.

The key difference between neural networks and the methods we saw in unit 2 is that, there, the mapping from x to $\phi(x)$ in not part of the data analysis, it is done ex-ante (even, although implicitly, with the choice of a particular kernel), and then we optimise once we chose ϕ.

In neural network instead the choice of the ϕ mapping is endogenous to the learning step, together with the choice of θ.

Note that we have a bit of a chicken and egg problem here, as, in order to do a good classification - understand and learn the parameters theta for the classification decision - we would need to know what that feature representation is. But, on the other hand, in order to understand what a good feature representation would be, we would need to know how that feature representation is exercised in the ultimate classification task.

So we need to learn ϕ and θ jointly, trying to adjust the feature representation together with how it is exercised towards the classification task.

Motivation to Neural Networks (an other summary):

So far, the ways we have performed non-linear classification involve either first mapping x explicitly into some feature vectors $\phi(x)$, whose coordinates involve non-linear functions of x, or in order to increase computational efficiency, rewriting the decision rule in terms of a chosen kernel, i.e. the dot product of feature vectors, and then using the training data to learn a transformed classification parameter.

However, in both cases, the feature vectors are chosen. They are not learned in order to improve performance of the classification problem at hand.

Neural networks, on the other hand, are models in which the feature representation is learned jointly with the classifier to improve classification performance.

8.4. Neural Network Units

Real neurons:

- take a signal in input using dendroids that connect the neuron to ~ 10^3 _ 10^4 other neurons
- the signal potential increases in the neuron's body
- once a threshold is reached, the neuron in turn emits a signal that, through its axon, reaches ~ 100 other neurons

Artificial neural networks:

- the parameter associated to each nodes of a layer (input coordinates) takes the role of weights trying to mimic the strength of the connection that propagates to the cell body;
- the response of a cell is in terms of a nonlinear transformation of the aggregated, weighted inputs, resulting in a single real number.

A bit of terminology:

- The individual coordinates are known as **nodes** or **units**.
- The **weights** are denoted with w instead of θ as we were used to.
- Each node's **aggregated input** is given by $z = \sum_{i=1}^{d} x_i w_i + w_0$ (or, in vector form, $z = \mathbf{x} \cdot \mathbf{w} + w_0$, with $z \in \mathbb{R}, \mathbf{x} \in \mathbb{R}^d, \mathbf{w} \in \mathbb{R}^d$)
- The output of the neuron is the result of a non-linear transformation of the aggregated input called **activation function** $f = f(z)$
- A **neural network unit** is a primitive neural network that consists of only the "input layer", and an output layer with only one output.

Common kind of activation functions:

- linear. Used typically at the very end, before measuring the loss of the predictor;
- relu ("rectified linear unit"): $f(z) = max\{0, z\}$
- tanh ("hyperbolic tangent"): it mimics the sine function but in a soft way: it is a sigmoid curve spanning from -1 (at $z = -\infty$) to +1 (at $z = +\infty$), with a value of 0 at $z = 0$. Its smoothness property is useful since we have to propagate the training signal through the network in order to adjust all the

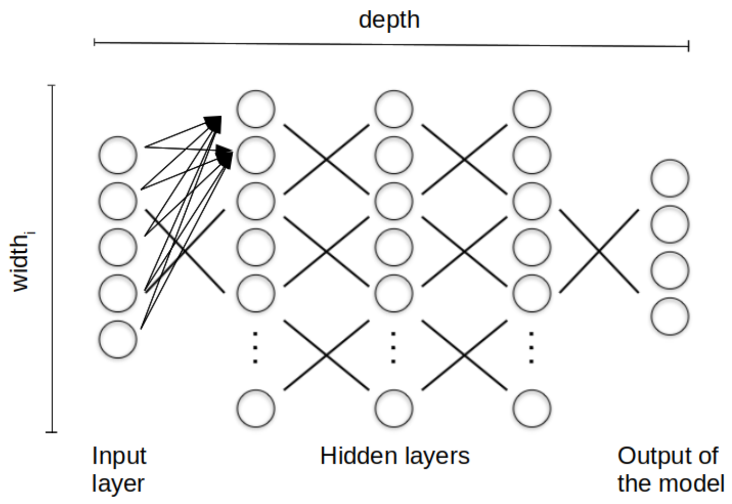
parameters in the model. $tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 1 - \frac{2}{e^{2z}+1}$. Note that $tanh(-z) = -tanh(z)$.

Our objective will be to learn the weights that make this node, in the context of the whole network, to then function appropriately, to behave well.

8.5. Introduction to Deep Neural Networks

Overall architecture

In **deep forward neural networks**, neural network units are arranged in **layers**, from the *input layer*, where each unit holds the input coordinate, through various *hidden layer* transformations, until the actual *output* of the model:



In this layerwise computation, each unit in a particular layer takes input from *all* the preceding layer units. And it has its own parameters that are adjusted to perform the overall computation. So parameters are different even between different units of the same layer. A deep (feedforward) neural network refers hence to a neural network that contains not only the input and output layers, but also hidden layers in between.

- **Width** (*of the layer*): number of units in that specific layer
- **Depth** (*of the architecture*): number of layers of the overall transformation before arriving to the final output

Deep neural networks

- loosely motivated by biological neurons, networks
- adjustable processing units (~ linear classifiers)
- **highly parallel** (important!), typically organized in layers
- deep = many transformations (layers) before output

For example the input could be an image, so the input vector is the individual pixel content, from which the first layer try to detect edges, then these are recombined into parts (in subsequent layers), objects and finally characterisation of a scene: edges -> simple parts-> parts -> objects -> scenes

One of the main advantages of deep neural networks is that in many cases, they can learn to extract very complex and sophisticated features from just the raw features presented to them as their input. For instance, in the context of image recognition, neural networks can extract the features that differentiate a cat from a dog based only on the raw pixel data presented to them from images.

The initial few layers of a neural networks typically capture the simpler and smaller features whereas the later layers use information from these low-level features to identify more complex and sophisticated features.

Note: it is interesting to note that a neural network can represent any given binary function.

Subject areas

Deep learning has overtaken a number of academic disciplines in just a few years:

- computer vision (e.g., image, scene analysis)
- natural language processing (e.g., machine translation)
- speech recognition
- computational biology, etc.

Key role in recent successes in corporate applications such as:

- self driving vehicles
- speech interfaces
- conversational agents, assistants (Alexa, Cortana, Siri, Google assistant,…)
- superhuman game playing

Many more underway (to perform prediction and/or control):

- personalized/automated medicine
- chemistry, robotics, materials science, etc.

Deep learning … why now?

1. Reason #1: lots of data

- many significant problems can only be solved at scale
- lots of data enable us to model very complex rich models solving more realistic tasks;

2. Reason #2: parallel kid computational resources (esp. GPUs, tensor processing units)

- platforms/systems that support running deep (machine) learning algorithms at scale in parallel

3. Reason #3: large models are actually easier to train

- contrary to small, rigid models, large, richer and flexible models can be successfully estimated even with simple gradient based learning algorithms like stochastic gradient descent.

4. Reason #4: flexible neural “lego pieces”

- common representations, diversity of architectural choices
- we can easily compose these models together to perform very interesting computations. In other words, they can serve as very flexible computational Lego pieces that can be adapted overall to perform a useful role as part of much larger, richer computational architectures.

Why #3 (large models are easier to learn) ?

We can think about the notions of width (number of units in a layer) and depth (number of layers). Small models (low width and low depth) are quite rigid and don’t allow for a good abstraction of reality i.e. learning the underlying structures based on observations. Large models can use more width and more depth to generate improved abstractions of reality i.e. improved learning.

See also the conclusion of the video on the next segment: “Introducing redundancy will make the optimization problem that we have to solve easier.”

8.6. Hidden Layer Models

Let’s now specifically consist a deep neural network consisting of the input layer $\mathbf{x}$, a single hidden layer $\mathbf{z}$ performing the aggregation $z_i = \sum_j x_j * w_{j,i} + w_{0,i}$ and using $tanh(\mathbf{z})$ as activation function f, and the output node with a linear activation function.

We can see each layer (one in this case) as a “box” that takes as input $\mathbf{x}$ and return output $\mathbf{f}$, mediated trough its weights $\mathbf{W}$, and the output layer as those box taking $\mathbf{f}$ as input and returning the final output f, mediated trough its weights $\mathbf{W}'$

And we are going to try to understand how this computation changes as a function of W and W'.

What these hidden units are actually doing ? Since they are like linear classifiers, they take linear combination of the inputs, pass through that nonlinear function, we can also visualize them as if they were linear classifiers with norm equal to $\mathbf{w}$.

The difference is that instead of having a binary output (like in $sign(\mathbf{w} \cdot \mathbf{x})$) we have now $f(\mathbf{w} \cdot \mathbf{x})$. f typically takes the form of $tanh(\mathbf{w} \cdot \mathbf{x})$ for the hidden layers that we will study, whose output range is (-1,+1). More the point is far from the boundary, more $tanh()$ move toward the extremes, with a speed that is proportional to the norm of $\mathbf{w}$.

If we have (as it is normally) multiple nodes per layer, we can thing on a series of linear classifiers on the same $depth_{i-1}$ space, each one identified by its norm $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_{depth_i}$.

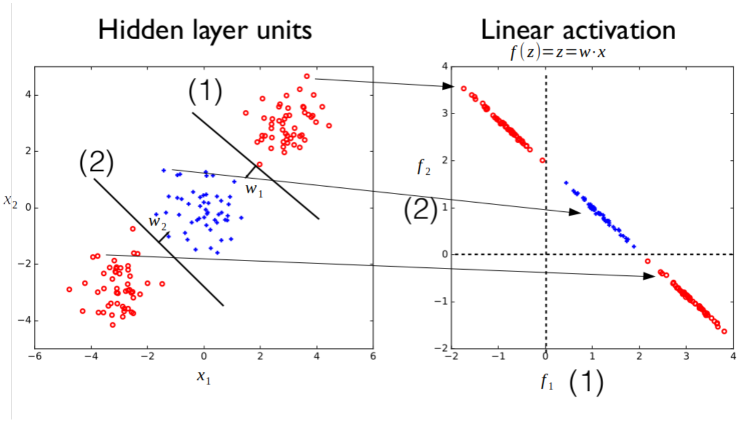
Given a neural network with one hidden layer for classification, we can view the hidden layer as a feature representation, and the output layer as a classifier using the learned feature representation.

There're also other parameters that will affect the learning process and the performance of the model, such as the learning rate and parameters that control the network architecture (e.g. number of hidden units/layers) etc. These are often called hyper-parameters.

Similar to the linear classifiers that we covered in previous lectures, we need to learn the parameters for the classifier. However, in this case we also learn the parameters that generate a representation for the data. The dimensions and the hyper-parameters are decided with the structure of the model and are not optimized directly during the learning process but can be chosen by performing a grid search with the evaluation data or by more advanced techniques (such as meta-learning).

2-D Example

Let's consider as example a case in 2-D where we have a cloud of negative points (red) in the bottom-left and top-right corners and a cloud of positive points (blue) in the center, like in the following chart (left side):

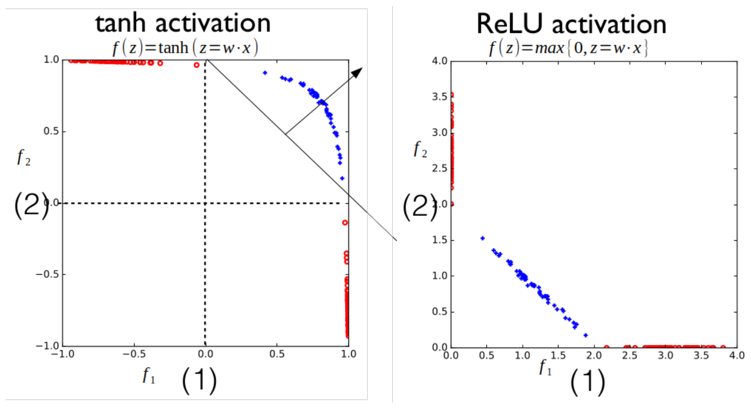


Such problem is clearly non linearly separable.

The chart on the right depicts the same points in the space resulting from the application of the two classifiers, using just a linear activation, i.e. the dot product between w and x.

The appearance that it draws exactly as a line derives from the fact that the two planes are parallel, but the general idea is that still we have a problem that is not separable, as any linear transformation of the feature space of a linearly in-separable classification problem would still continue to remain linearly inseparable.

However, when we use $tanh(x)$ as activation function we obtain the output as depicted in the following chart (left), where the problem now is clearly linearly separable.



This is really where the power of the hidden layer lies. It gives us a transformation of the input signal into a representation that makes the problem easier to solve.

Finally we can try the ReLU activation ($f(z) = max\{0, z\}$): in this case the output is not actually strictly linearly separable.

So this highlights the difficulty of learning these models. It’s not always the case that the same non-linear transformation casts them as linearly separable.

However if we flip the planes directions, both the tanh and the ReLU activation results in outputs that become linearly separable (for the ReLU, all positive points got mapped to the (0,0) point).

What if we chose the two planes as random (i.e. $\mathbf{w}_1$ and $\mathbf{w}_2$ are random) ? The resulting output would likely not be linearly separable. However if we introduce redundancy, e.g. using 10 hidden units for an original two dimensional problem, the problem would likely become linearly separable even if these 10 planes are chosen at random. Notice this is quite similar to the systematic expansion that we did earlier, in terms of polynomial features.

So introducing redundancy here is actually helpful. And we’ll see how this is helpful also when we are actually learning these hidden unit representations from data. Introducing redundancy will make the optimization problem that we have to solve easier.

Summary

- Units in neural networks are linear classifiers, just with different output non-linearity
- The units in feed-forward neural networks are arranged in layers (input, one or plus hidden, output)
- By learning the parameters associated with the hidden layer units, we learn how to represent examples (as hidden layer activations)
- The representations in neural networks are learned directly to facilitate the end-to-end task
- A simple classifier (output unit) suffices to solve complex classification tasks if it operates on the hidden layer representations

The outward layer is their prediction that we actually want. And the role of the hidden layers is really to adjust their transformation, adjust their computation in such a way that the output layer will have an easier task to solve the problem.

The next lecture will deal with actually learning these representations together with the final classifier.

Lecture 9. Feedforward Neural Networks, Back Propagation, and Stochastic Gradient Descent (SGD)

9.1. Objectives

At the end of this lecture, you will be able to

- Write down recursive relations with back-propagation algorithm to compute the gradient of the loss function with respect to the weight parameters.
- Use the stochastic descent algorithm to train a feedforward neural network.
- Understand that it is not guaranteed to reach global (only local) optimum with SGD to minimize the training loss.
- Recognize when a network has overcapacity .

9.2. Back-propagation Algorithm

We start now to consider how to learn from data (feedforward) neural network, that is estimate its weights.

We recall that feed-forward neural networks, with multiple hidden layers mediating the calculation from the input to the output, are complicated models that are trying to capture the representation of the examples towards the output unit in such a way as to facilitate the actual prediction task.

It is this representation learning part – we're learning the feature representation as well as how to make use of it – that makes the learning problem difficult. But it turns out that a simple stochastic gradient descent algorithm actually succeeds in finding typically a good solution to the parameters, provided that we give the model a little bit of overcapacity. The main algorithmic question, then, is how to actually evaluate that gradient, the derivative of the Loss with respect to the parameters. And that can be computed efficiently using so-called back propagation algorithm.

Given an input X, a neural network characterised by the overall weight set W (so that its output, a scalar here for simplicity, is $f(X;W)$), and the "correct" target vector Y, the task in the training step is find the W that minimise a given loss function $\mathcal{L}(f(X;W),Y)$. We do that by computing the derivative of the loss function for each weight $w^l_{i,j}$ applied by the j-th unit at each l-th layer in relation to the output of the i-th node at the previous $l-1$ layer: $\frac{\partial \mathcal{L}(f(X;W),Y)}{\partial w_{i,j}}$.

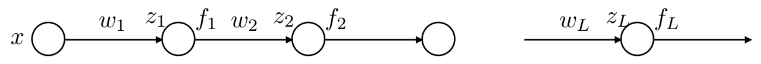
Then we simply apply the SDG algorithm in relation to this $w^l_{i,j}$:

$$w_{i,j}^l \leftarrow w_{i,j}^l - \eta * \frac{\partial \mathcal{L}(f(X;W),Y)}{\partial w_{i,j}}$$

The question turns now on how do we evaluate such gradient, as the mapping from the weight to the final output of the network can be very complicated (so much for the weights in the first layers!).

This computation can be efficiently done using a so called **back propagation algorithm** that essentially exploits the chain rule.

Let's take as example a deep neural network with a single unit per node, with both input and outputs as scalars, as in the following diagram:



Let's also assume that the activation function is $\tanh(z)$, also in the last layer (in reality the last unit is often a linear function, so that the prediction is in $\mathbb{R}$ and not just in $(-1,+1)$), that there is no offset parameter and that the specific loss function for each individual example is $Loss = \frac{1}{2}(y-f_L)^2$.

Then, for such network, we can write $z_1 = xw_1$ and, more in general, for $i = 2,\ldots,L$: $z_i = f_{i-1}w_i$ where $f_{i-1} = f(z_{i-1})$.

So, specifically, $f_1 = tanh(z_1) = tanh(xw_1)$, $f_2 = tanh(z_2) = tanh(f_1 w_2)$,...

In order to find $\frac{\partial Loss}{\partial w_i}$ we can use the chain rule by start evaluating the derivative of the loss function, then evaluate the last layer, and then eventually go backward until the first layer:

$$\frac{\partial Loss}{\partial w_1} = \frac{\partial f_1}{\partial w_1} * \frac{\partial f_2}{\partial f_1} * \frac{\partial f_3}{\partial f_2} * \ldots * \frac{\partial f_L}{\partial f_{L-1}} * \frac{\partial Loss}{\partial f_L}$$

$$\frac{\partial Loss}{\partial w_1} = [(1-tanh^2(xw_1))x] * [(1-tanh^2(f_1 w_2))w_2] * [(1-tanh^2(f_2 w_3))w_3] * \ldots [(1$$

Now based on the nature the calculator, the fact that we evaluate the loss at the very output, then multiply by these Jacobians also highlights how this can go wrong. Imagine if these Jacobians here, the value is the derivatives of the layer-wise mappings, are very small. Then the gradient vanishes very quickly as the depth of the architecture increases. If these derivatives are large, then the gradients can also explode.

So there are issues that we need to deal with when the architecture is deep.

9.3. Training Models with 1 Hidden Layer, Overcapacity, and Convergence Guarantees

Using the SGD, the average hinge loss evolves at each iterations (or epochs), where the algorithm runs through all the training examples (in sample order), performing a stochastic gradient descent update Typically, after a few runs over the training examples, the network actually succeeds in finding a solution that has zero hinge loss.

When the problem is however complex, a neural network with just the capacity in terms of number of nodes that would be theoretically enough to find a separable solution (classify all the example correctly) may not actually arrive to such optimal classification. More in general, for multi-layer neural networks, stochastic gradient descent (SGD) is not guaranteed to reach a global optimum (but they can find a locally optimal solution, which is typically quite good).

We can facilitate the optimization of these architectures by giving them **overcapacity** (increasing the number of nodes in the layer(s)), making them a little bit more complicated than they need to be to actually solve the task.

Using overcapacity however can lead to artefacts in the classifiers. To limit these artefacts and have good models even with overcapacity one can use two tricks:

1. Consider random initialisation of the parameters (rather than start from zero). And as we learn and arrive at a perfect solution in terms of end-to-end mapping from inputs to outputs, then in that solution, not all the hidden units need to be doing something useful, so long as many of them do. The randomization inherent in the initialization creates some smoothness. And we end up with a smooth decision boundary even when we have given the model quite a bit of overcapacity.
2. Use the ReLU activation function ($max(0, z)$) rather than $tanh(z)$. ReLU is cheap to evaluate , works well with sparsity and make parameters easier to be estimated in large models.

We will later talk about regularization – how to actually squeeze the capacity of these models a little bit, while in terms of units, giving them overcapacity.

Summary

- Neural networks can be learned with SGD similarly to linear classifiers
- The derivatives necessary for SGD can be evaluated effectively via back-propagation
- Multi-layer neural network models are complicated. We are no longer guaranteed to reach global (only local) optimum with SGD
- Larger models tend to be easier to learn because their units only need to be adjusted so that they are, collectively, sufficient to solve the task

Lecture 10. Recurrent Neural Networks 1

10.1. Objective

Introduction to recurrent neural networks (RNNs)

At the end of this lecture, you will be able to:

- Know the difference between feed-forward and recurrent neural networks(RNNs).
- Understand the role of gating and memory cells in long-short term memory (LSTM).

- Understand the process of encoding of RNNs in modeling sequences.

10.2. Introduction to Recurrent Neural Networks

In this and in the next lecture we will use neural networks to model sequences, using so called **recurrent neural networks** (*RNN*).

This lecture introduces the topic: the problem of modelling sequences, what are RNN, how they relate to the feedforward neural network we saw in the previous lectures and how to *encode* sequences into vector representations. Next lecture will focus on how to *decode* such vectors so that we can use them to predict properties of the sequences, or what comes next in the sequence.

Exchange rate example

Let’s consider a time serie of the exchange rate between US Dollar and the Euro, with an objective of predict its value in the future.

We already saw how to solve this kind of problem with linear predictors or feedforward neural networks.

In both case the first task is to compile a feature vector, for example of the values of the exchange rate at various times, for example, at time $t-1$ to $t-4$ for a prediction at time t.

As we have long time-serie available we can use some of the observation as training, some as validation and some as test (ideally by random sampling).

Language completion example

In a similar way we can see a text as a sequence of words, and try to predict the next world based on the previous words, for example using the previous two words, where each of them is coded as a 1 in a sparse array of all the possible words (à la bag of words approach).

Limitations of these approaches

While we could use linear classifiers or feedforward neural networks to predict sequences we are still left with the problem of how much “history” look at in the creation of the feature vector and the fact that this history length may be variable, for example there may be words at the beginning of the sentence that are quite relevant for predicting what happens towards the end, and we would have to somehow retain that information in the feature representation that we are using for predicting what happens next.

Recurrent Neural Networks can be used in place of feedforwrd neural networks to learn not only the weight given the feature vectors, but also how to encode in the first instance the history into the feature vector.

10.3. Why we need RNNs

The way we chose how to encode data in feature vectors depends on the task we need. For example, sentiment analysis, language translation, and next word suggestion all requires a different feature representation as they focus on different parts of the sentence: while sentiment analysis focuses on the holistic meaning of a sentence, translation or next word suggestion focuses instead more on individual words.

While in feed-forward networks we have to manually engineer how history is mapped to a feature vector (representation) for the specific task at hand, using RNN's this task is also part of the learning process and hence automatised.

Note that very different types of objects can be encoded in feature vectors, like images, events, words or videos, and once they are encoded, all these different kind of objects can be used together.

In other words, RNNs can not only process single data points (such as images), but also entire sequences of data (such as speech or video).

While this lecture deals with **encoding**, the mapping of a sequence to a feature vector, the next lecture deals with **decoding**, the mapping of a feature vector to a sequence.

10.4. Encoding with RNN

While in feedforward neural network the input is only at the beginning of the chain and the parameter matrix W is different at each layer, in recurrent neural networks the "external input" arrives at each layer and contribute to the argument of the activation function together with the flow of information coming from the previous layer (*recurrent* refers to this state information that, properly transformed, flows across the various layers).

One simple implementation of each layer transformation (that here we can see it as an "update") is hence (omitting the offset parameters):

$s_t = \tanh(W^{s,s} s_{t-1} + W^{s,x} x_t)$

Where:

- s_{t-1} is a $m \times 1$ vector of the old "context" or "state" (the data coming from the previous layer)
- $W^{s,s}$ is a $m \times m$ matrix of the weights associated to the existing state, whose role is to deciding what part of the previous information should be keep (and note that this is not changing in each layer.). Can also be interpreted as giving how the state would evolve in absence of any new information.
- x_t is a $d \times 1$ feature representation of the new information (e.g. a new word)
- $W^{s,x}$ is a $m \times m$ weights whose role is deciding how to take into account the new information, so that the result of wx multiplication is specific to each new information arriving;
- $tanh(\cdot)$ is the activation function (to be applied elementwise)
- s_t is a $m \times 1$ vector of the new "context" or "state" (the updated state with the new information taken into account)

RNN have hence a number of layers equal to the data in the sequence, like the words in a sentence. So there is a single, evolving, NN for the whole sequence rather than a different NN for each element of the sequence. The initial state (S_0) is a vector of m zeros.

Note that this parametric approach let the way we introduce new data ($W^{s,x}$) to adjust to the way we use the network, i.e. to be learned according to the specific problem on hand…

In other words, we can adjust those parameters to make this representation of a sequence appropriate for the task that we are trying to solve.

10.5. Gating and LSTM

Learning RNNs

Learning a RNN is similar to learning a feedforward neural network: the first task is to define a Loss function and then find the weights that minimise it, for example using a SGD algorithm where the gradient with respect to the weights is computed using back-propagation. The fact that the parameters are shared in RNN's means that we add the contribution of each of the suggested modifications for parameters at each of these positions where the transformation is flagged. One problem of simple RNN models is that the gradient can vanish or explode. Here even more than in feedforward NN, as the sequences can be quite long and we apply this transformation repeatedly. In real cases, the design of the transformation can be improved to counter this issue.

Simple gated RNN

Further, in simple RNN the state information is *always* updated with new data, so far away information is easily "forget". It is often helpful to retain (or learn to retain) some control over what is written over and what is incorporated as new information. We can apply this control over what we overwrite and what instead we retain from the old state when meet new data using a form of RNN called **gated recursive neural networks**. Gated networks adds *gates* controlling the flow of information. Its simplest implementation can be written as:

$$g_t = \text{sigmoid}(W^{g,s}s_{t-1} + W^{g,x}x_t) = \frac{1}{1+e^{-(W^{g,s}s_{t-1}+W^{g,x}x_t)}}$$

$$s_t = (1-g_t) \odot s_{t-1} + g_t \odot \tanh(W^{s,s}s_{t-1} + W^{s,x}x_t)$$

Where the first equation defines a **gate** responsible to filter in the new information (trough its own parameters to be learn as well) and results in a continuous value between 0 and 1.

The second equation then uses the gate to define, for each individual value of the state vector (the $\odot$ symbol stands for element-wise multiplication), how much to retain and how much to update with the new information. For example, if $g_t[2]$ (the second element of the gate at the t transformation) is 0 (an extreme value!), it means we keep in that transformation the old value of the state for the second element, ignoring the transformation deriving from the new information.

Long Short Term Memory neural networks

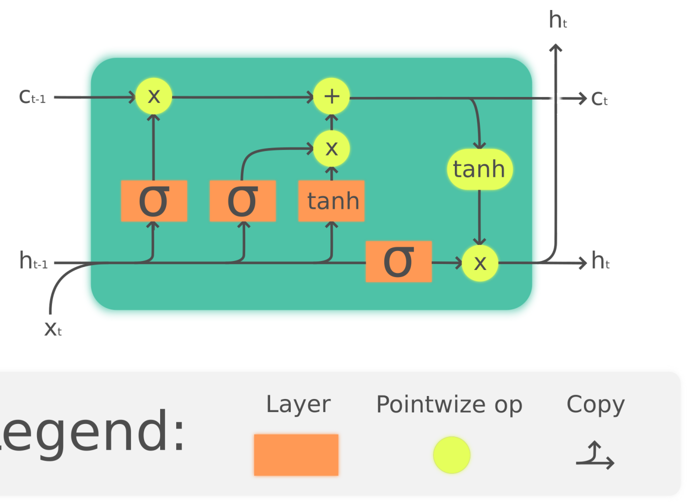
Real in-use gated RNN are even more complicated. In particular, **Long Short Term Memory** (recursive neural) networks (shortered as LSTM) are well-suited to classifying, processing and making predictions based on time series data, since there can be lags of unknown duration between important events in a time series and have the following gates defined:

$$
\begin{aligned}
f_t &= sigmoid(W^{f,h} h_{t-1} + W^{f,x} x_t) && \text{forget gate} \\
i_t &= sigmoid(W^{i,h} h_{t-1} + W^{i,x} x_t) && \text{input gate} \\
o_t &= sigmoid(W^{o,h} h_{t-1} + W^{o,x} x_t) && \text{output gate} \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tanh(W^{c,h} h_{t-1} + W^{c,x} x_t) && \text{memory cell} \\
h_t &= o_t \odot tanh(c_t) && \text{visible state}
\end{aligned}
$$

The input, forget, and output gates control respectively how to read information into the memory cell, how to forget information that we've had previously, and how to output information from the memory cell into a visible form.

The "state" is now represented collectively by the memory cell c_t 'sometimes indicated as *long-term memory*) and its "visible" state h_t (sometimes indicated as *working memory* or *hidden state*).

LSTM cells have hence the following diagram:



The memory cell update is helpful to retain information over a longer sequences. But we keep his memory cell hidden and instead only reveal the visible portion of this tape. And it is this h_t then at the end of the whole sequence applying this box along the sequence that we will use as the vector representation for the sequence.

On the LSTM topic one could also look at these external resources:

- http://blog.echen.me/2017/05/30/exploring-lstms/
- https://colah.github.io/posts/2015-08-Understanding-LSTMs/
- https://machinelearningmastery.com/handle-long-sequences-long-short-term-memory-recurrent-neural-networks/

Key things

- Neural networks for sequences: encoding
- RNNs, unfolded
    - state evolution, gates
    - relation to feed-forward neural networks
    - back-propagation (conceptually)
- Issues: vanishing/exploding gradient
- LSTM (operationally)

- RNN’s turn sequences into vectors (encoding)
- They can be understood as feed-forward neural networks whose architecture has changed from one sequence to another
- Can be learned with back-propagation of the error signal like for feedforward NN
- They too suffer of vanishing or exploding gradient issues
- Specific architectures such as the LSTM maintain a better control over the information that’s retained or updated along the sequence, and they are therefore easier to train

Lecture 11. Recurrent Neural Networks 2

11.1. Objective

From Markov model to recurrent neural networks (RNNs)

- Formulate, estimate and sample sequences from Markov models.
- Understand the relation between RNNs and Markov model for generating sequences.
- Understand the process of decoding of RNN in generating sequences.

11.2. Markov Models

Outline

- Modeling sequences: language models
    - Markov models
    - as neural networks
    - hidden state, Recurrent Neural Networks (RNNs)
- Example: decoding images into sentences

While in the last lesson we saw how to transform a sentence into a vector, in a parametrized way that can be optimized for what we want the vector to do, today we’re going to be talking about how to generate sequences using recurrent neural networks (decoding). For example, in the translation domain, how to unravel the output vector as a sentence in an other language.

Markov models

One way to implement prediction in a sequence is to just first define probabilities for any possible combination looking at existing data (for example, in a next word prediction - “language modelling”, look at all two-word combinations in a serie of texts) and then, once we observe a case, sample the next element from this conditional (discrete) probability distribution. This is a first order Markov model.

Markov textual bigram model exemple

In this example we want to predict the next word based exclusively on the previous one.

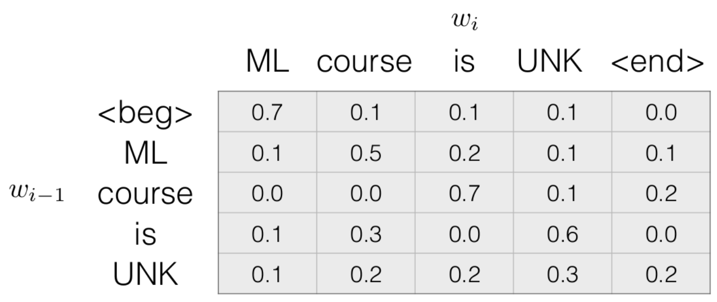
We first learn the probability of any pair of words from data (“corpus”). For practical reasons, we consider a vocabulary V of the n more frequent words (and symbols), labelling all the others as `UNK` (a catch-all for “unknown”). To this vocabulary we also add two special symbols `<beg>` and `<end>` to mark respectively the beginning and the ending of the sentence. So a pair `(<beg>,w)` would represent a word $w \in V$ starting the sentence and `(w,<end>)` would represent the world w ending the sentence.

We can now estimate the probability for each words w and $w' \in V$ that w' follows w, that is the conditional probability that next word is w' given the previous one was w, by a normalised counting of the successive occurrences of the pair (w, w') (matching statistics):

$$\hat{P}(w'|w) = \frac{count(w,w')}{\sum_{w_i \in V} count(w,w_i)}$$

(we could be a bit smarter and consider at the denominator only the set of pairs whose first element is w rather than the whole $V \in n^2$)

For a bigram we would obtain a probability table like the following one:



		w_i				
		ML	course	is	UNK	<end>
	<beg>	0.7	0.1	0.1	0.1	0.0
	ML	0.1	0.5	0.2	0.1	0.1
w_{i-1}	course	0.0	0.0	0.7	0.1	0.2
	is	0.1	0.3	0.0	0.6	0.0
	UNK	0.1	0.2	0.2	0.3	0.2

At this point we can generate a sentence by each time sampling from the conditional probability mass function of the last observed word, starting with `<beg>` (first row in the table) and until we sample a `<end>`.

We can use the table also to define the probability of any given sentence by just using the probability multiplication rule. The probability of any N words sentence (including `<beg>` and `<end>`) is then $P = \prod_{i=2}^{N} P(w_i|w_{i-1})$. For example, given the above table, the probability of the sentence “Course is great” would be $0.1 * 0.7 * 0.6 * 0.2$.

Note that, using the counting approach, the model maximise the probability that the generated sequences correspond to the observed sequences in the corpus, i.e. counting corresponds here to the Maximum Likelihood Estimation of the conditional probabilities. Also, the same approach can be used to model words characted by character rather than sentences world by world.

Outline detailed

In this segment we considered language modelling, using the very simple sequence model called Markov model. In the next segments of this lecture we're going to turn those Markov models into a neural network models, first as feed-forward neural network models. And then we will add a hidden state and turn them into recurrent neural network models. And finally, we'll just consider briefly an example of unraveling a vector representation, in this case, coming from an image to a sequence.

11.3. Markov Models to Feedforward Neural Nets

We can represent the first order Markov model described in the previous segment as a feed-forward neural network,

We start by a one-hot encoding of the words, i.e. each input word would activate one unique node on the input layer of the network ($\mathbf{x}$). In other words, if the vocabulary is composed of K words, the input layer of the neural network has width K, and it is filed all with zeros except for the specific word encoded as 1.

We want as output of the neural network the PMF conditional to that specific word in the input. So the output layer has too one unit for each possible word, returning each node the probability that the next word is that specific one given that the previous one was those encoded in x: $P_k = P(w_i = k|w_{i-1})$

Given the weights of this neural network W (not to be confused with the words w), the argument of the activation function of each node of the output layer is $z_k = \mathbf{x} \cdot \mathbf{W}_k + W_{0,k}$.

These z are real numbers. To transform in probabilities (all positive, sum equal to 1) we use as activation function the non-linear Softmax function:

$P_k = \frac{e^{Z_k}}{\sum_{j=1}^{K} e^{Z_j}}$

(and the output layer is then called **softmax output layer**).

Advantages of neural network representation

Frexibility

We can use as input of the neural network a vector x coposed of the one-hot econding of the previous world, *plus* the vector of the one-hot encoding of the second previous word, obtaining the probability that the next word is w_k conditional to the two preceding words (roughly similar to a second order Markov model).

Further, we could also insert an hidden layer in between th input and output layers, in order to look at more complex combinations of the preceding two words, in terms of how they are mapped to the probability values over the next word.

Parsimony

For the tri-gram model above, the neural network would need a weight matrix W of $2K \times K$ parameters, i.e; a total og $2K^2$ parameters.

Using a second order Markov model would require instead K^3 parameters: we take two preceding words, and for any such combination, we predict a probability of what the next word is. So if we have here the possible words roughly speaking, then each combination of preceding words, the square root of them, and then for each of those combinations, we have to have a probability value of each of the next words. So we will look at that times the number of parameters in the tri-gram model. So roughly, the number of words to the power of 3.

As a result, the neural network representation for a tri-gram model is not as general, but it is much more feasible in terms of the number of parameters that we have to estimate. But we can always increase the complexity of the model in the neural network representation by adding a hidden layer.

11.4. RNN Deeper Dive

We can now translate the feedforward rural network in a Recursive Neural Network that accepts a *sequence* of words as input and can account for a variable history in making predictionsfor the next word rather than on a fixed number of elements (as 1 or 2 in bigrams and trigrams respectively).

The framework is similar to the RNN we saw in the previous lesson, with a state (initially set to $\mathbf{0} \in R^K$) that is updated, at each new information, with a function of the previous state and the new observed word (typically with something like $s_t = tanh(W^{s,s} s_{t-1} + W^{s,w} x_t)$). Note that x_t is the one hot encoding of the new observed word.

The difference is that in addition, at each step, we have also an output that transforms the state in probability (representing the new, conditional PMF): $p_t = softmax(W^0 s_t)$.

Note that the state here retains information of all the history of the sentence, hence the probability is conditional to all the previous history in the sentence.

In other words, while $W^{s,s}$ and $W^{s,w}$ role is to select and encode the relevant features from the previous history and the new data respectively, W^0 role is to extract the relevant features from the memorised state with the aim of making a prediction.

Finally we can have more complex structures, like LSTM networks, with forget, input and output gates and the state divided in a memory cell and a visible state. Also in this case we would however have a further transformation that output the conditional PMF as function of the visible state ($p_t = softmax(W^0 h_t)$).

Note that the training phase is done computing the average loss at the level of sentences, i.e. our labels are the full sentences, and are these that are compared in the loss function with those obtained by sampling the PMFs resulting from the RNN. While in *training* we use the true words specified as input for the next time step, in *testing* instead we let the rural network to predict the sentence on its own, using the sampled output at one time step as the input for the next step.

11.5. RNN Decoding

The question is how to use now these (trained) RNN models to generate a sentence from a given encoded state (e.g. in testing)?

The only difference is that the initial state is not a vector of zero, but the “encoded sentence”. We just start with that state and the `<beg>` symbol as x and then let the RNN produce a PDF, sample from that and use that sampled data as the new x for the next step and so on until we sample an `<end>`.

We don’t just have to take a vector from a sentence and translate it into another sentence. We can take a vector of representation of an image that we have learned or are learning in the same process and translate that into a sentence.

So we could take an image, translate into a state using a convolutional neural network and then this state is the initial state of an other neural network predicting the caption sentence associated to that image (e.g. “A person riding a motorcycle on a dirt road”)

Key summary

- Markov models for sequences
  - how to formulate, estimate, sample sequences from
- RNNs for generating (decoding) sequences
  - relation to Markov models (how to translate Markov models into RNN)
  - evolving hidden state
  - sampling from the RNN at each point
- Decoding vectors into sequences (once we have this architecture that can generate sequences, such as sentences, we can start any vector coming from a sentence, image, or other context and unravel it as a sequence, e.g. as a natural language sentence.)

Homework 4

Lecture 12. Convolutional Neural Networks (CNNs)

12.1. Objectives

At the end of this lecture, you will be able to

- Know the differences between feed-forward and Convolutional neural networks (CNNs).
- Implement the key parts in the CNNs, including *convolution*, *max pooling* units.
- Determine the dimension of each channel in different layers with a given CNNs.

12.2. Convolutional Neural Networks

CNNs Outline:

- why not use unstructured feed-forward models?
- key parts: convolution, pooling
- examples

The problem: image classification, i.e. multi-way classification of images mapping to a specific category (e.g. x is a given image, the label is “dog” or “car”).

Why we don’t use “normal” feed-forward neural networks ?

1. It would need too many parameters. If an image is mid-size resolution 1000×1000, each layer would need a weight matrix connecting all these 10^6 pixel in input with 10^6 pixel in output, i.e. 10^12 weights
2. Local learning (in the image) would not extend to global learning. If we train the network to recognise cars, and it happens that our training photos have the cars all in the bottom half, then the network would not recognise a car in the top half, as these would activate different neurons.

We solve these problems employing specialised network architectures called convolution neural network.

In these networks the layer l is obtained by operating over the image at layer $l-1$ a small **filter** (or **kernel**) that is slid across the image with a step of 1 (typically) or more pixels at the time. The step is called **stride**, while the whole process of sliding the filter can be mathematically seen as a **convolution**.

So, while we slide the filter, at each location of the filter, the output is composed of the dot product between the values of the filter and the corresponding location in the image (both vectorised), where the values of the filters are the weigths that we want to learn, and they remain constant across the sliding. If our filter is a 10×10 matrix, we have only 10 weights to learn by layer (plus one for the offset). As in feedforward neural network then the dot product undergo an activation function, here typically the ReLU function ($max(0, x)$).

For example, given an image $x = \begin{bmatrix} 1 & 1 & 2 & 1 & 1 \\ 3 & 1 & 4 & 1 & 1 \\ 1 & 3 & 1 & 2 & 2 \\ 1 & 2 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 & 1 \end{bmatrix}$ and filter weights $w = \begin{bmatrix} 1 & -2 & 0 \\ 1 & 0 & 1 \\ -1 & 1 & 0 \end{bmatrix}$, then the output of the filter z would be $\begin{bmatrix} 8 & -3 & 6 \\ 4 & -3 & 5 \\ -3 & 5 & -2 \end{bmatrix}$. For example, the element of this matrix $z_{2,3} = 5$ is the result of the sum of the scalar multiplication between $x' = \begin{bmatrix} 4 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 1 & 1 \end{bmatrix}$ and w.

Finally, the output of the layer would be (using ReLU) $\begin{bmatrix} 8 & 0 & 6 \\ 4 & 0 & 5 \\ 0 & 5 & 0 \end{bmatrix}$.

You can notice that we obtain a dimensionality reduction applying the filter, that depends on the dimension of the filter and the stride (sliding step). In order to avoid this a padding of zeros can be applied to the image in order to keep the same dimensions in the output (in the above example a padding of one zeros on both sides - and both dimensions - would suffice).

Because the weight of the filters are the same, it doesn't really matter where the object is learned, in which part of the image. The lecture explains this with a mushroom example: ff the mushroom is in a different place, in a feed-forward neural network the weight matrix parameters at that location need to learn to recognize the mushroom anew. With convolutional layers, we have translational invariance as the same filter is passed over the entire image. Therefore, it will detect the mushroom regardless of location.

Still, it is often convenient to operate some **data augmentation** to the training set, that is to add slightly modified images (rotated, mirrored...) in order to improve this translational invariance.

An further way to improve translational invariance, but also have some dimensionality reduction, it called **pooling** and is adding a layer with a filter whose output is the max of the corresponding area in the input. Note that this layer would have no weights! With pooling we contribute to start separating what is in the image from where it is in the image, that is pooling does a fine-scale, local translational invariance, while convolution does more a large-scale one.

Keeping the output of the above example as input, a pooling layer with a 2×2 filter and a stride of 1 would result in $\begin{bmatrix} 8 & 6 \\ 5 & 5 \end{bmatrix}$.

In a typical CNN, these convolutional and pooling layers are repeated several times, where the initial few layers typically would capture the simpler and smaller features, whereas the later layers would use information from these low-level features to identify more complex and sophisticated features, like characterisations of a scene. The learned weights would hence specialise across the layers in a sequence like edges -> simple parts-> parts -> objects -> scenes.

Concerning the topic of CNN, see also the superb lecture of Andrej Karpathy on YouTube (here or here).

12.3. CNN - Continued

CNN’s are architectures combine these type of layers successively in a variety of different ways.

Typically one single layer is formed by applying multiple filter, not just one. This is because we want to learn different kind of features… for example one filter will activated to catch vertical lines in the image, an other obliques ones… and maybe an other different colours. And by the way the image and the filter have normally a further dimensions to account for colour (typically of size 3):

96 convolutional filters on the first layer (filters are of size 11x11x3, applied across input images of size 224x224x3). They all have been learned from random initialisation.

So in each layer we are going to map the original image into multiple feature maps where each feature map is generated by a little weight matrix, the filter, that defines the little classifier that’s run through the original image to get the associated feature map. Each of these feature maps defines a channel for information.

I can then combine these convolution, looking for features, and pooling, compressing the image a little bit, forgetting the information of where things are, but maintaining what is there.

These layers are finally followed by some “normal”, “fully connected” layers (*à la* feed-forward neural network) and a final softmax layer indicating the probability that each image represents one of the possible categories (there could be thousand of them).

The best network implementation are tested in so called “competitions”; like the yearly ImageNet context.

Note that we can train this networks exactly like for feedformard NN, defining a loss function and finding the weights that minimise the loss function. In particular we can apply the stochastic gradient descendent algorithm (with a few tricks based on getting pairs of image and the corresponding label), where the gradient with respect to the various parameters (weights) is obtained by backpropagation.

Take home

- Understand what a convolution is

- Understand the pooling, that tries to generate a slightly more compressed image forgetting where things are, but maintaining information about what's there, what was activated.

Recitation: Convolution/Cross Correlation:

Convolution (of continuous functions): $(f*g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$

Cross-correlation: $(f*g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t+\tau)d\tau$ (i.e. the g function is not reversed)

In neural network we use the cross-correlation rather than the convolution. Indeed, in such context f is he data signal and g is the filter. But the filter needs to be learn, so if it expressed straight or reversed doesn't really matter, so we just use the cross-correlation and save one operation.

1-D Discrete version

Cross correlation of discrete functions: $(f*g)(t) := \sum_{\tau=-\infty}^{\infty} f(\tau)g(t+\tau)$

The cross-correlation $h = (f*g)(t)$ for discrete functions can be pictured in the following scheme, where the output is the cross product of the values of f and g in the same column, and where out of domain values are padded with zeros.

```
f:           1 2 3
...
       0| 1 2
h(t=0) 2|   1 2
h(t=1) 5|     1 2
h(t=2) 8|       1 2
h(t=3) 3|         1 2
h(t=4) 0|           1 2
...
```

2-D Discrete version

In 2-D the cross correlation becomes:
$(f*g)(x,y) := \sum_{\tau_1=-\infty}^{\infty} \sum_{\tau_2=-\infty}^{\infty} f(\tau_1,\tau_2)g(\tau_1+x,\tau_2+y)$.

Graphically it is the sliding of the filter (first row, left to right, second row, left to right, …) that we saw in the previous segment.

Project 3: Digit recognition (Part 2)

[MITx 6.86x Notes Index]