1 Convolutional Neural Networks

1.1 Overview

- Architecture of a traditional CNN – Convolutional neural networks, also known as CNNs, are a specific type of neural networks that are generally composed of the following layers:

```
Input image  Convolutions  Pooling  Fully Connected
```

The convolution layer and the pooling layer can be fine-tuned with respect to hyperparameters that are described in the next sections.

1.2 Types of layer

- Convolutional layer (CONV) – The convolution layer (CONV) uses filters that perform convolution operations as it is scanning the input \( I \) with respect to its dimensions. Its hyperparameters include the filter size \( F \) and stride \( S \). The resulting output \( O \) is called feature map or activation map.

Remark: the convolution step can be generalized to the 1D and 3D cases as well.

- Pooling (POOL) – The pooling layer (POOL) is a downsampling operation, typically applied after a convolution layer, which does some spatial invariance. In particular, max and average pooling are special kinds of pooling where the maximum and average value is taken, respectively.
## Max pooling vs. Average pooling

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Max pooling</th>
<th>Average pooling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each pooling operation selects the maximum value of the current view</td>
<td>Each pooling operation averages the values of the current view</td>
<td></td>
</tr>
</tbody>
</table>

### Illustration

- **Max pooling**: Keeps the maximum value of the current view.
- **Average pooling**: Averages the values of the current view.

### Comments

- **Max pooling**
  - Preserves detected features
  - Most commonly used
  - Downsamples feature map
  - Used in LeNet

- **Average pooling**
  - No padding
  - Drops last convolution if dimensions do not match
  - Padding such that feature map size has size $\lceil \frac{I}{S} \rceil$
  - Output size is mathematically convenient
  - Also called ‘half’ padding
  - Maximum padding such that end convolutions are applied on the limits of the input
  - Filter ‘sees’ the input end-to-end

### Fully Connected (FC)

- The fully connected layer (FC) operates on a flattened input where each input is connected to all neurons. If present, FC layers are usually found towards the end of CNN architectures and can be used to optimize objectives such as class scores.

### Filter hyperparameters

The convolution layer contains filters for which it is important to know the meaning behind its hyperparameters.

#### Dimensions of a filter

A filter of size $F \times F$ applied to an input containing $C$ channels is a $F \times F \times C$ volume that performs convolutions on an input of size $I \times I \times C$ and produces an output feature map (also called activation map) of size $O \times O \times 1$.

**Remark**: the application of $K$ filters of size $F \times F$ results in an output feature map of size $O \times O \times K$.

#### Stride

For a convolutional or a pooling operation, the stride $S$ denotes the number of pixels by which the window moves after each operation.

**Remark**: often times, $P_{start} = P_{end} = P$, in which case we can replace $P_{start} + P_{end}$ by $2P$ in the formula above.

### Tuning hyperparameters

#### Parameter compatibility in convolution layer

- By noting $I$ the length of the input volume size, $F$ the length of the filter, $P$ the amount of zero padding, $S$ the stride, then the output size $O$ of the feature map along that dimension is given by:

$$O = \frac{I - F + P_{start} + P_{end}}{S} + 1$$
Understanding the complexity of the model – In order to assess the complexity of a model, it is often useful to determine the number of parameters that its architecture will have. In a given layer of a convolutional neural network, it is done as follows:

<table>
<thead>
<tr>
<th>Illustration</th>
<th>CONV</th>
<th>POOL</th>
<th>FC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$F \times K$</td>
<td>$F$</td>
<td>$N_{in} \times N_{out}$</td>
</tr>
</tbody>
</table>

**Input size**
- $I \times I \times C$

**Output size**
- $O \times O \times K$

**Number of parameters**
- $(F \times F \times C + 1) \cdot K$
- 0
- $(N_{in} + 1) \times N_{out}$

**Remarks**
- One bias parameter per filter
- Input is flattened
- The receptive field at layer $k$ is $2C$
- Pooling operation done channel-wise
- In most cases, $S = F$
- In most cases, $S < F$
- A common choice for $K$ is 2

Receptive field – The receptive field at layer $k$ is the area denoted $R_k \times R_k$ of the input that each pixel of the $k$-th activation map can ’see’. By calling $F_j$ the filter size of layer $j$ and $S_j$ the stride value of layer $i$ and with the convention $S_0 = 1$, the receptive field at layer $k$ can be computed with the formula:

$$R_k = 1 + \sum_{j=1}^{k} (F_j - 1) \prod_{i=0}^{j-1} S_i$$

In the example below, we have $F_1 = F_2 = 3$ and $S_1 = S_2 = 1$, which gives $R_2 = 1 + 2 \cdot 1 + 2 \cdot 1 + 2 \cdot 1 = 5$.

1.5 Commonly used activation functions

Rectified Linear Unit – The rectified linear unit layer (ReLU) is an activation function $g$ that is used on all elements of the volume. It aims at introducing non-linearities to the network. Its variants are summarized in the table below:

<table>
<thead>
<tr>
<th>ReLU</th>
<th>Leaky ReLU</th>
<th>ELU</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g(z) = \max(0,z)$</td>
<td>$g(z) = \max(\epsilon z,z)$ where $\epsilon \ll 1$</td>
<td>$g(z) = \max(\alpha(e^z - 1),z)$ where $\alpha \ll 1$</td>
</tr>
</tbody>
</table>

Non-linearity complexities
- Biologically interpretable
- Addresses dying ReLU issue for negative values
- Differentiable everywhere

1.6 Object detection

Types of models – There are 3 main types of object recognition algorithms, for which the nature of what is predicted is different. They are described in the table below:

<table>
<thead>
<tr>
<th>Image classification</th>
<th>Classification w. localization</th>
<th>Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional CNN</td>
<td>Simplified YOLO, R-CNN</td>
<td>YOLO, R-CNN</td>
</tr>
</tbody>
</table>

- Classifies a picture
- Predicts probability of object
- Predicts object in a picture
- Predicts probability of object and where it is located
- Detects up to several objects in a picture
- Predicts probabilities of objects and where they are located

Detection – In the context of object detection, different methods are used depending on whether we just want to locate the object or detect a more complex shape in the image. The two main ones are summed up in the table below:
Bounding box detection

- Detects the part of the image where the object is located
- More granular

Box of center \((b_x, b_y)\), height \(b_h\) and width \(b_w\)

Reference points \((l_{1x}, l_{1y}), \ldots, (l_{nx}, l_{ny})\)

Intersection over Union

Intersection over Union, also known as IoU, is a function that quantifies how correctly positioned a predicted bounding box \(B_p\) is over the actual bounding box \(B_a\). It is defined as:

\[
\text{IoU}(B_p, B_a) = \frac{B_p \cap B_a}{B_p \cup B_a}
\]

Remark: we always have \(\text{IoU} \in [0,1]\). By convention, a predicted bounding box \(B_p\) is considered as being reasonably good if \(\text{IoU}(B_p, B_a) \geq 0.5\).

Anchor boxes

Anchor boxing is a technique used to predict overlapping bounding boxes. In practice, the network is allowed to predict more than one box simultaneously, where each box prediction is constrained to have a given set of geometrical properties. For instance, the first prediction can potentially be a rectangular box of a given form, while the second will be another rectangular box of a different geometrical form.

Non-max suppression

The non-max suppression technique aims at removing duplicate overlapping bounding boxes of a same object by selecting the most representative ones. After having removed all boxes having a probability prediction lower than 0.6, the following steps are repeated while there are boxes remaining:

- Step 1: Pick the box with the largest prediction probability.
- Step 2: Discard any box having an IoU \(\geq 0.5\) with the previous box.

YOLO – You Only Look Once (YOLO) is an object detection algorithm that performs the following steps:

- Step 1: Divide the input image into a \(G \times G\) grid.
- Step 2: For each grid cell, run a CNN that predicts \(y\) of the following form:

\[
y = \begin{bmatrix} p_c, b_x, b_y, b_h, b_w, c_1, c_2, \ldots, c_p \end{bmatrix}^T \in \mathbb{R}^{G \times G \times k \times (5+p)\text{ repeated } k \text{ times}}
\]

where \(p_c\) is the probability of detecting an object, \(b_x, b_y, b_h, b_w\) are the properties of the detected bounding box, \(c_1, \ldots, c_p\) is a one-hot representation of which of the \(p\) classes were detected, and \(k\) is the number of anchor boxes.
- Step 3: Run the non-max suppression algorithm to remove any potential duplicate overlapping bounding boxes.

Remark: when \(p_c = 0\), then the network does not detect any object. In that case, the corresponding predictions \(b_x, \ldots, c_p\) have to be ignored.

R-CNN – Region with Convolutional Neural Networks (R-CNN) is an object detection algorithm that first segments the image to find potential relevant bounding boxes and then run the detection algorithm to find most probable objects in those bounding boxes.

Remark: although the original algorithm is computationally expensive and slow, newer architectures enabled the algorithm to run faster, such as Fast R-CNN and Faster R-CNN.
1.6.1 Face verification and recognition

**Types of models** – Two main types of models are summed up in the table below:

<table>
<thead>
<tr>
<th>Face verification</th>
<th>Face recognition</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Is this the correct person?</td>
<td>- Is this one of the $K$ persons in the database?</td>
</tr>
<tr>
<td>- One-to-one lookup</td>
<td>- One-to-many lookup</td>
</tr>
</tbody>
</table>

- **Query**
  - [Image 111x381 to 207x466]
  - Is this the correct person?
  - Is this one of the $K$ persons in the database?

- **Reference**
  - [Image 93x111 to 207x210]
  - Is this one of the $K$ persons in the database?

- **Database**
  - [Image 219x111 to 334x210]
  + Is this the correct person?
  - Is this one of the $K$ persons in the database?

- **Activation** – In a given layer $l$, the activation is noted $a^{[l]}$ and is of dimensions $n_h \times n_w \times n_c$.

- **Siamese Network** – Siamese Networks aim at learning how to encode images to then quantify how different two images are. The similarity function applied to two images is often noted $d(x^{(1)}, x^{(2)})$.

**One Shot Learning** – One Shot Learning is a face verification algorithm that uses a limited training set to learn a similarity function that quantifies how different two given images are. The similarity function applied to two images is often noted $d(x^{(1)}, x^{(2)})$.

- **Face verification**
  - Two main types of models are summed up in the table below:

- **Face recognition**
  - Two main types of models are summed up in the table below:

- **Query**
  - [Image 111x381 to 207x466]
  - Is this the correct person?
  - Is this one of the $K$ persons in the database?

- **Reference**
  - [Image 93x111 to 207x210]
  - Is this one of the $K$ persons in the database?

- **Database**
  - [Image 219x111 to 334x210]
  + Is this the correct person?
  - Is this one of the $K$ persons in the database?

- **One-to-one lookup**
- **One-to-many lookup**
- **Activation** – In a given layer $l$, the activation is noted $a^{[l]}$ and is of dimensions $n_h \times n_w \times n_c$.

- **One Shot Learning** – One Shot Learning is a face verification algorithm that uses a limited training set to learn a similarity function that quantifies how different two given images are. The similarity function applied to two images is often noted $d(x^{(1)}, x^{(2)})$.

1.6.2 Neural style transfer

**Motivation** – The goal of neural style transfer is to generate an image $G$ based on a given content $C$ and a given style $S$.

- **Activation** – In a given layer $l$, the activation is noted $a^{[l]}$ and is of dimensions $n_h \times n_w \times n_c$.

- **Siamese Network** – Siamese Networks aim at learning how to encode images to then quantify how different two images are. The similarity function applied to two images is often noted $d(x^{(1)}, x^{(2)})$.

**One Shot Learning** – One Shot Learning is a face verification algorithm that uses a limited training set to learn a similarity function that quantifies how different two given images are. The similarity function applied to two images is often noted $d(x^{(1)}, x^{(2)})$.

**Siamese Network** – Siamese Networks aim at learning how to encode images to then quantify how different two images are. The similarity function applied to two images is often noted $d(x^{(1)}, x^{(2)})$.

**One Shot Learning** – One Shot Learning is a face verification algorithm that uses a limited training set to learn a similarity function that quantifies how different two given images are. The similarity function applied to two images is often noted $d(x^{(1)}, x^{(2)})$.

**One Shot Learning** – One Shot Learning is a face verification algorithm that uses a limited training set to learn a similarity function that quantifies how different two given images are. The similarity function applied to two images is often noted $d(x^{(1)}, x^{(2)})$.

**Triplet loss** – The triplet loss $\ell$ is a loss function computed on the embedding representation of a triplet of images $A$ (anchor), $P$ (positive) and $N$ (negative). The anchor and the positive example belong to the same class, while the negative example to another one. By calling $\alpha \in \mathbb{R^+}$ the margin parameter, this loss is defined as follows:

\[
\ell(A, P, N) = \max(d(A, P) - d(A, N) + \alpha, 0)
\]

\[
\ell(A, P, N) = 0 \quad \text{if } \ell(A, P, N) > 0
\]

1.6.3 Architectures using computational tricks

**Generative Adversarial Network** – Generative adversarial networks, also known as GANs, are composed of a generative and a discriminative model, where the generative model aims at generating the most truthful output that will be fed into the discriminative which aims at differentiating the generated and true image.
Remark: use cases using variants of GANs include text to image, music generation and synthesis.

ResNet – The Residual Network architecture (also called ResNet) uses residual blocks with a high number of layers meant to decrease the training error. The residual block has the following characterizing equation:

\[ a^{[l+2]} = g(a^{[l]} + z^{[l+2]}) \]

Inception Network – This architecture uses inception modules and aims at giving a try at different convolutions in order to increase its performance. In particular, it uses the 1x1 convolution trick to lower the burden of computation.

* * *

2 Recurrent Neural Networks

2.1 Overview

Architecture of a traditional RNN – Recurrent neural networks, also known as RNNs, are a class of neural networks that allow previous outputs to be used as inputs while having hidden states. They are typically as follows:

For each timestep \( t \), the activation \( a^{<t>} \) and the output \( y^{<t>} \) are expressed as follows:

\[ a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a) \]
\[ y^{<t>} = g_2(W_{ya}a^{<t>} + b_y) \]

where \( W_{ax}, W_{aa}, W_{ya}, b_a, b_y \) are coefficients that are shared temporally and \( g_1, g_2 \) activation functions.

The pros and cons of a typical RNN architecture are summed up in the table below:

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Drawbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Possibility of processing input of any length</td>
<td>- Computation being slow</td>
</tr>
<tr>
<td>- Model size not increasing with size of input</td>
<td>- Difficulty of accessing information</td>
</tr>
<tr>
<td>- Computation takes into account</td>
<td>- from a long time ago</td>
</tr>
<tr>
<td>historical information</td>
<td>- Cannot consider any future input</td>
</tr>
<tr>
<td>- Weights are shared across time</td>
<td>- for the current state</td>
</tr>
</tbody>
</table>

Applications of RNNs – RNN models are mostly used in the fields of natural language processing and speech recognition. The different applications are summed up in the table below:
<table>
<thead>
<tr>
<th>Type of RNN</th>
<th>Illustration</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>One-to-one</td>
<td><img src="image1" alt="Illustration" /></td>
<td>Traditional neural network</td>
</tr>
<tr>
<td>$T_x = T_y = 1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>One-to-many</td>
<td><img src="image2" alt="Illustration" /></td>
<td>Music generation</td>
</tr>
<tr>
<td>$T_x = 1, T_y &gt; 1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Many-to-one</td>
<td><img src="image3" alt="Illustration" /></td>
<td>Sentiment classification</td>
</tr>
<tr>
<td>$T_x &gt; 1, T_y = 1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Many-to-many</td>
<td><img src="image4" alt="Illustration" /></td>
<td>Name entity recognition</td>
</tr>
<tr>
<td>$T_x = T_y$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Many-to-many</td>
<td><img src="image5" alt="Illustration" /></td>
<td>Machine translation</td>
</tr>
<tr>
<td>$T_x \neq T_y$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**2.2 Handling long term dependencies**

- **Commonly used activation functions** - The most common activation functions used in RNN modules are described below:

<table>
<thead>
<tr>
<th>Function</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sigmoid</td>
<td>$g(z) = \frac{1}{1 + e^{-z}}$</td>
</tr>
<tr>
<td>Tanh</td>
<td>$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$</td>
</tr>
<tr>
<td>RELU</td>
<td>$g(z) = \max(0, z)$</td>
</tr>
</tbody>
</table>

- **Vanishing/exploding gradient** - The vanishing and exploding gradient phenomena are often encountered in the context of RNNs. The reason why they happen is that it is difficult to capture long term dependencies because of multiplicative gradient that can be exponentially decreasing/increasing with respect to the number of layers.

- **Gradient clipping** - It is a technique used to cope with the exploding gradient problem sometimes encountered when performing backpropagation. By capping the maximum value for the gradient, this phenomenon is controlled in practice.

**Loss function** - In the case of a recurrent neural network, the loss function $\mathcal{L}$ of all time steps is defined based on the loss at every time step as follows:

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}(\hat{y}^{\langle t \rangle}, y^{\langle t \rangle})$$

**Backpropagation through time** - Backpropagation is done at each point in time. At timestep $T$, the derivative of the loss $\mathcal{L}$ with respect to weight matrix $W$ is expressed as follows:

$$\frac{\partial \mathcal{L}(T)}{\partial W} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}(T)}{\partial W} \bigg|_{(t)}$$

**Types of gates** - In order to remedy the vanishing gradient problem, specific gates are used in some types of RNNs and usually have a well-defined purpose. They are usually noted $\Gamma$ and are equal to:

$$\Gamma = \sigma(Wx^{\langle t \rangle} + Ua^{\langle t-1 \rangle} + b)$$

$\Gamma$, $W$, $U$, $b$ are coefficients specific to the gate and $\sigma$ is the sigmoid function. The main ones are summed up in the table below.
GRU/LSTM – Gated Recurrent Unit (GRU) and Long Short-Term Memory units (LSTM) deal with the vanishing gradient problem encountered by traditional RNNs, with LSTM being a generalization of GRU. Below is a table summing up the characterizing equations of each architecture:

<table>
<thead>
<tr>
<th>Type of gate</th>
<th>Role</th>
<th>Used in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update gate $\Gamma_u$</td>
<td>How much past should matter now?</td>
<td>GRU, LSTM</td>
</tr>
<tr>
<td>Relevance gate $\Gamma_r$</td>
<td>Drop previous information?</td>
<td>GRU, LSTM</td>
</tr>
<tr>
<td>Forget gate $\Gamma_f$</td>
<td>Erase a cell or not?</td>
<td>LSTM</td>
</tr>
<tr>
<td>Output gate $\Gamma_o$</td>
<td>How much to reveal of a cell?</td>
<td>LSTM</td>
</tr>
</tbody>
</table>

\[ \tilde{c}^{<t>} = \tanh(W_c[\Gamma_r \star a^{<t-1>}, x^{<t>}] + b_c) \]

\[ c^{<t>} = \Gamma_u \star \tilde{c}^{<t>} + (1 - \Gamma_u) \star c^{<t-1>} \]

\[ a^{<t>} = \Gamma_o \star c^{<t>} \]

\[ \text{Dependencies} \]

Remark: the sign $\star$ denotes the element-wise multiplication between two vectors.

Variants of RNNs – The table below sums up the other commonly used RNN architectures:

<table>
<thead>
<tr>
<th>Bidirectional (BRNN)</th>
<th>Deep (DRNN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

2.3 Learning word representation

In this section, we note $V$ the vocabulary and $|V|$ its size.

2.3.1 Motivation and notations

Representation techniques – The two main ways of representing words are summed up in the table below:

<table>
<thead>
<tr>
<th>1-hot representation</th>
<th>Word embedding</th>
</tr>
</thead>
<tbody>
<tr>
<td>teddy bear</td>
<td>soft</td>
</tr>
<tr>
<td>book</td>
<td>soft</td>
</tr>
<tr>
<td>- Noted $o_w$</td>
<td>- Noted $e_w$</td>
</tr>
<tr>
<td>- Naive approach, no similarity information</td>
<td>- Takes into account words similarity</td>
</tr>
</tbody>
</table>

Embedding matrix – For a given word $w$, the embedding matrix $E$ is a matrix that maps its 1-hot representation $o_w$ to its embedding $e_w$ as follows:

\[ e_w = E o_w \]

Remark: learning the embedding matrix can be done using target/context likelihood models.

2.3.2 Word embeddings

Word2vec – Word2vec is a framework aimed at learning word embeddings by estimating the likelihood that a given word is surrounded by other words. Popular models include skip-gram, negative sampling and CBOW.

Skip-gram – The skip-gram word2vec model is a supervised learning task that learns word embeddings by assessing the likelihood of any given target word $t$ happening with a context word $c$. By noting $\theta_t$ a parameter associated with $t$, the probability $P(t|c)$ is given by:
Negative sampling – It is a set of binary classifiers using logistic regressions that aim at assessing how a given context and a given target words are likely to appear simultaneously, with the models being trained on sets of $k$ negative examples and 1 positive example. Given a context word $c$ and a target word $t$, the prediction is expressed by:

$$P(y = 1|c,t) = \sigma(\theta^T c_t)$$

Remark: this method is less computationally expensive than the skip-gram model.

GloVe – The GloVe model, short for global vectors for word representation, is a word embedding technique that uses a co-occurrence matrix $G$ where each $X_{i,j}$ denotes the number of times that a target $i$ occurred with a context $j$. Its cost function $J$ is as follows:

$$J(\theta) = \frac{1}{2} \sum_{i,j=1}^{||V||} f(X_{i,j})(\theta_i^T e_j + b_i + b_j' - \log(X_{i,j}))^2$$

here $f$ is a weighting function such that $X_{i,j} = 0 \Rightarrow f(X_{i,j}) = 0$.

Given the symmetry that $e$ and $\theta$ play in this model, the final word embedding $e_w^{\text{(final)}}$ is given by:

$$e_w^{\text{(final)}} = \frac{e_w + \theta_w}{2}$$

Remark: the individual components of the learned word embeddings are not necessarily interpretable.

Comparing words

Cosine similarity – The cosine similarity between words $w_1$ and $w_2$ is expressed as follows:

$$\text{similarity} = \frac{w_1 \cdot w_2}{||w_1|| \cdot ||w_2||} = \cos(\theta)$$

Remark: $\theta$ is the angle between words $w_1$ and $w_2$.

t-SNE – t-SNE (t-distributed Stochastic Neighbor Embedding) is a technique aimed at reducing high-dimensional embeddings into a lower dimensional space. In practice, it is commonly used to visualize word vectors in the 2D space.

Language model

Overview – A language model aims at estimating the probability of a sentence $P(y)$.

$n$-gram model – This model is a naive approach aiming at quantifying the probability that an expression appears in a corpus by counting its number of appearance in the training data.

Perplexity – Language models are commonly assessed using the perplexity metric, also known as PP, which can be interpreted as the inverse probability of the dataset normalized by the number of words $T$. The perplexity is such that the lower, the better and is defined as follows:

$$PP = \prod_{t=1}^{T} \left( \frac{1}{\sum_{j=1}^{||V||} P(y_j|y_{<t})} \right)^{1/T}$$

Remark: $PP$ is commonly used in t-SNE.

Machine translation

Overview – A machine translation model is similar to a language model except it has an encoder network placed before. For this reason, it is sometimes referred as a conditional language model. The goal is to find a sentence $y$ such that:

$$y = \arg \max_{y^{<1},...,y^{<T_y}} P(y^{<1},...,y^{<T_y}|x)$$

Beam search – It is a heuristic search algorithm used in machine translation and speech recognition to find the likeliest sentence $y$ given an input $x$.

- Step 1: Find top $B$ likely words $y^{<1}$
- Step 2: Compute conditional probabilities $y^{<k}|x,y^{<1},...,y^{<k-1}$
- Step 3: Keep top $B$ combinations $x,y^{<1},...,y^{<k}$
Remark: if the beam width is set to 1, then this is equivalent to a naive greedy search.

- **Beam width** – The beam width $B$ is a parameter for beam search. Large values of $B$ yield better results but with slower performance and increased memory. Small values of $B$ lead to worse results but is less computationally intensive. A standard value for $B$ is around 10.

- **Length normalization** – In order to improve numerical stability, beam search is usually applied on the following normalized objective, often called the normalized log-likelihood objective, defined as:

$$
\text{Objective} = \frac{1}{T_y} \sum_{t=1}^{T_y} \log \left[ p(y^{<t}>|x,y^{<1>},...,y^{<t-1>}) \right]
$$

Remark: the parameter $\alpha$ can be seen as a softener, and its value is usually between 0.5 and 1.

- **Error analysis** – When obtaining a predicted translation $\hat{y}$ that is bad, one can wonder why we did not get a good translation $y^*$ by performing the following error analysis:

| Case          | $P(y^*|x) > P(\hat{y}|x)$ | $P(y^*|x) \leq P(\hat{y}|x)$ |
|---------------|---------------------------|-------------------------------|
| Root cause    | Beam search faulty        | RNN faulty                    |
| Remedies      | Increase beam width       | - Try different architecture  |
|               |                           | - Regularize                  |
|               |                           | - Get more data               |

- **Bleu score** – The bilingual evaluation understudy (bleu) score quantifies how good a machine translation is by computing a similarity score based on n-gram precision. It is defined as follows:

$$
\text{bleu score} = \exp \left( \frac{1}{n} \sum_{k=1}^{n} p_k \right)
$$

where $p_n$ is the bleu score on n-gram only defined as follows:

$$
p_n = \frac{\sum_{\text{count}_{\text{clip}}(\text{n-gram})}}{\sum_{\text{n-gram} \in \hat{y}} \text{count}(\text{n-gram})}
$$

Remark: a brevity penalty may be applied to short predicted translations to prevent an artificially inflated bleu score.

- **Attention weight** – The amount of attention that the output $y^{<t>}$ should pay to the activation $a^{<t',t>}$ is given by $\alpha^{<t,t'>}$ computed as follows:

$$
\alpha^{<t,t'>} = \frac{\exp(\epsilon^{<t,t'>})}{\sum_{t''=1}^{T_x} \exp(\epsilon^{<t,t''>})}
$$

Remark: computation complexity is quadratic with respect to $T_x$.

2.7 Attention

- **Attention model** – This model allows an RNN to pay attention to specific parts of the input that is considered as being important, which improves the performance of the resulting model in practice. By noting $\alpha^{<t,t'>}$ the amount of attention that the output $y^{<t>}$ should pay to the activation $a^{<t',t>}$ and $c^{<t,t>}$ the context at time $t$, we have:

$$
c^{<t,t>} = \sum_{t'} \alpha^{<t,t'>} a^{<t'>} \quad \text{with} \quad \sum_{t'} \alpha^{<t,t'>} = 1
$$

Remark: the attention scores are commonly used in image captioning and machine translation.

Stanford University 10 Winter 2019
3 Deep Learning Tips and Tricks

3.1 Data processing

Data augmentation – Deep learning models usually need a lot of data to be properly trained. It is often useful to get more data from the existing ones using data augmentation techniques. The main ones are summed up in the table below. More precisely, given the following input image, here are the techniques that we can apply:

<table>
<thead>
<tr>
<th>Original</th>
<th>Flip</th>
<th>Rotation</th>
<th>Random crop</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Image without any modification</td>
<td>- Flipped with respect to an axis for which the meaning of the image is preserved</td>
<td>- Rotation with a slight angle</td>
<td>- Random focus on one part of the image</td>
</tr>
<tr>
<td>- Simulates incorrect horizon calibration</td>
<td>- Several random crops can be done in a row</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Color shift</th>
<th>Noise addition</th>
<th>Information loss</th>
<th>Contrast change</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Nuances of RGB is slightly changed</td>
<td>- Addition of noise</td>
<td>- Parts of image ignored</td>
<td>- Luminosity changes</td>
</tr>
<tr>
<td>- Captures noise that can occur with light exposure</td>
<td>- More tolerance to quality variation of inputs</td>
<td>- Mimics potential loss of parts of image</td>
<td>- Controls difference in exposition due to time of day</td>
</tr>
</tbody>
</table>

Batch normalization – It is a step of hyperparameter $\gamma, \beta$ that normalizes the batch $\{x_i\}$. By noting $\mu_B, \sigma^2_B$ the mean and variance of that we want to correct to the batch, it is done as follows:

$$x_i \leftarrow \gamma \frac{x_i - \mu_B}{\sqrt{\sigma^2_B + \epsilon}} + \beta$$

It is usually done after a fully connected/convolutional layer and before a non-linearity layer and aims at allowing higher learning rates and reducing the strong dependence on initialization.

3.2 Training a neural network

3.2.1 Definitions

Epoch – In the context of training a model, epoch is a term used to refer to one iteration where the model sees the whole training set to update its weights.

Mini-batch gradient descent – During the training phase, updating weights is usually not based on the whole training set at once due to computation complexities or one data point due to noise issues. Instead, the update step is done on mini-batches, where the number of data points in a batch is a hyperparameter that we can tune.

Loss function – In order to quantify how a given model performs, the loss function $L$ is usually used to evaluate to what extent the actual outputs $y$ are correctly predicted by the model outputs $z$.

Cross-entropy loss – In the context of binary classification in neural networks, the cross-entropy loss $L(z, y)$ is commonly used and is defined as follows:

$$L(z, y) = - y \log(z) + (1 - y) \log(1 - z)$$

3.2.2 Finding optimal weights

Backpropagation – Backpropagation is a method to update the weights in the neural network by taking into account the actual output and the desired output. The derivative with respect to each weight $w$ is computed using the chain rule.

$$\frac{\partial L}{\partial f(x)} \times \frac{\partial f(x)}{\partial x}$$

Using this method, each weight is updated with the rule:

$$w \leftarrow w - \alpha \frac{\partial L(z, y)}{\partial w}$$

Updating weights – In a neural network, weights are updated as follows:

1. Step 1: Take a batch of training data and perform forward propagation to compute the loss.
2. Step 2: Backpropagate the loss to get the gradient of the loss with respect to each weight.
3. Step 3: Use the gradients to update the weights of the network.
3.3 Parameter tuning

3.3.1 Weights initialization

☐ Xavier initialization – Instead of initializing the weights in a purely random manner, Xavier initialization enables to have initial weights that take into account characteristics that are unique to the architecture.

☐ Transfer learning – Training a deep learning model requires a lot of data and more importantly a lot of time. It is often useful to take advantage of pre-trained weights on huge datasets that took days/weeks to train, and leverage it towards our use case. Depending on how much data we have at hand, here are the different ways to leverage this:

<table>
<thead>
<tr>
<th>Training size</th>
<th>Illustration</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td><img src="image" alt="Small Network" /></td>
<td>Freezes all layers, trains weights on softmax</td>
</tr>
<tr>
<td>Medium</td>
<td><img src="image" alt="Medium Network" /></td>
<td>Freezes most layers, trains weights on last layers and softmax</td>
</tr>
<tr>
<td>Large</td>
<td><img src="image" alt="Large Network" /></td>
<td>Trains weights on layers and softmax by initializing weights on pre-trained ones</td>
</tr>
</tbody>
</table>

3.3.2 Optimizing convergence

☐ Learning rate – The learning rate, often noted $\alpha$ or sometimes $\eta$, indicates at which pace the weights get updated. It can be fixed or adaptively changed. The current most popular method is called Adam, which is a method that adapts the learning rate.

☐ Adaptive learning rates – Letting the learning rate vary when training a model can reduce the training time and improve the numerical optimal solution. While Adam optimizer is the most commonly used technique, others can also be useful. They are summed up in the table below:

<table>
<thead>
<tr>
<th>Method</th>
<th>Explanation</th>
<th>Update of $w$</th>
<th>Update of $b$</th>
</tr>
</thead>
</table>
| Momentum | - Dampens oscillations  
- Improvement to SGD  
- 2 parameters to tune | $w \leftarrow w - \alpha v_{dw}$ | $b \leftarrow b - \alpha v_{db}$ |
| RMSprop | - Root Mean Square propagation  
- Speeds up learning algorithm by controlling oscillations | $w \leftarrow w - \alpha \frac{dw}{\sqrt{s_{dw}}}$ | $b \leftarrow b - \alpha \frac{db}{\sqrt{s_{db}}}$ |
| Adam | - Adaptive Moment estimation  
- Most popular method  
- 4 parameters to tune | $w \leftarrow w - \alpha \frac{v_{dw}}{\sqrt{s_{dw}} + \epsilon}$ | $b \leftarrow b - \alpha \frac{v_{db}}{\sqrt{s_{db}} + \epsilon}$ |

Remark: other methods include Adadelta, Adagrad and SGD.

3.4 Regularization

☐ Dropout – Dropout is a technique used in neural networks to prevent overfitting the training data by dropping out neurons with probability $p > 0$. It forces the model to avoid relying too much on particular sets of features.

Remark: most deep learning frameworks parametrize dropout through the 'keep' parameter $1 - p$.

☐ Weight regularization – In order to make sure that the weights are not too large and that the model is not overfitting the training set, regularization techniques are usually performed on the model weights. The main ones are summed up in the table below:

<table>
<thead>
<tr>
<th>Technique</th>
<th>Explanation</th>
<th>LASSO</th>
<th>Ridge</th>
<th>Elastic Net</th>
</tr>
</thead>
</table>
| - Shrinks coefficients to 0  
- Good for variable selection | | $|\theta|_1 \leq 1$ | $|\theta|_2 \leq 1$ | $(1 - \alpha)|\theta|_1 + \alpha|\theta|_2^2 \leq 1$ |
| - Makes coefficients smaller | $\lambda \in \mathbb{R}$ | $\lambda \in \mathbb{R}$ | $\lambda \in \mathbb{R}, \alpha \in [0,1]$ |
| - Tradeoff between variable selection and small coefficients | $\lambda \in \mathbb{R}$ | $\lambda \in \mathbb{R}$ | $\lambda \in \mathbb{R}, \alpha \in [0,1]$ |
Early stopping – This regularization technique stops the training process as soon as the validation loss reaches a plateau or starts to increase.

![Graph showing early stopping](image)

3.5 Good practices

Overfitting small batch – When debugging a model, it is often useful to make quick tests to see if there is any major issue with the architecture of the model itself. In particular, in order to make sure that the model can be properly trained, a mini-batch is passed inside the network to see if it can overfit on it. If it cannot, it means that the model is either too complex or not complex enough to even overfit on a small batch, let alone a normal-sized training set.

Gradient checking – Gradient checking is a method used during the implementation of the backward pass of a neural network. It compares the value of the analytical gradient to the numerical gradient at given points and plays the role of a sanity-check for correctness.

<table>
<thead>
<tr>
<th>Numerical gradient</th>
<th>Analytical gradient</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formula</td>
<td></td>
</tr>
<tr>
<td>( \frac{df}{dx}(x) \approx \frac{f(x+h) - f(x-h)}{2h} )</td>
<td>( \frac{df}{dx}(x) = f'(x) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Comments</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>- Expensive; loss has to be computed two times per dimension</td>
<td>- 'Exact' result</td>
</tr>
<tr>
<td>- Used to verify correctness of analytical implementation</td>
<td>- Direct computation</td>
</tr>
<tr>
<td>- Trade-off in choosing ( h ) not too small (numerical instability) nor too large (poor gradient approx.)</td>
<td>- Used in the final implementation</td>
</tr>
</tbody>
</table>

* * *