Learn a method in 5 minutes: Focal Loss

• Hypothesis:
  • We can modify the loss function to pay more attention to samples that are more difficult to learn.

• Method:
  Taking $p_t$ as $p_t = \begin{cases} p, & \text{if } y = 1 \\ 1 - p, & \text{otherwise} \end{cases}$

We modify CE from:

$$CE(p_t) = -\log(p_t)$$

To:

$$FL(p_t) = -(1 - p_t)^\gamma \log(p_t)$$
Feed-forward through convolution

\[ a^l_i = \sigma(net^l_i) \]

\[ net^l_i = \sum_{j=1}^{F} w_j \cdot a^{l-1}_{i+j-1} \]

For example:
\[ net^l_1 = w_1 a^{l-1}_1 + w_2 a^{l-1}_2 + w_3 a^{l-1}_3 \]
Backpropagation through convolution

Feedforward:
\[ a_i^l = \sigma(\text{net}_i^l) \]
\[ \text{net}_i^l = \sum_{j=1}^{F} w_j \cdot a_{i+j-1}^{l-1} \]

Gradient wrt. weights:
\[
\frac{\partial L}{\partial w_k} = \frac{\partial L}{\partial a_1^l} \frac{\partial a_1^l}{\partial w_k} + \frac{\partial L}{\partial a_2^l} \frac{\partial a_2^l}{\partial w_k} + \ldots
\]
\[
= \sum_i \frac{\partial L}{\partial a_i^l} \frac{\partial a_i^l}{\partial w_k} + \sum_i \frac{\partial L}{\partial a_i^l} \frac{\partial a_i^l}{\partial \text{net}_i^l} \frac{\partial \text{net}_i^l}{\partial w_k}
\]
Backpropagation through convolution

Feedforward:
\[ a^l_i = \sigma(\text{net}^l_i) \]
\[ \text{net}^l_i = \sum_{j=1}^F w_j \cdot a^{l-1}_{i+j-1} \]

Gradient wrt. input layer:
\[ \frac{\partial L}{\partial a^{l-1}_3} = ? \]
\[ = \frac{\partial L}{\partial a^l_1} \frac{\partial a^l_1}{\partial \text{net}^l_1} + \frac{\partial L}{\partial a^l_2} \frac{\partial a^l_2}{\partial \text{net}^l_2} + \frac{\partial L}{\partial a^l_3} \frac{\partial a^l_3}{\partial \text{net}^l_3} \]
\[ = \frac{\partial L}{\partial \text{net}^l_1} w_3 + \frac{\partial L}{\partial \text{net}^l_2} w_2 + \frac{\partial L}{\partial \text{net}^l_3} w_1 \]

In general:
\[ \frac{\partial L}{\partial a^{l-1}_i} = \sum_{j=1}^F \frac{\partial L}{\partial \text{net}^{l-j}_i} w_j \]
A Blueprint for CNNs

INPUT \rightarrow [[\text{CONV} \rightarrow \text{RELU}]^*N \rightarrow \text{POOL}]^*M \rightarrow [\text{FC} \rightarrow \text{RELU}]^*K \rightarrow \text{FC}

where the * indicates repetition, and the \text{POOL}^? indicates an optional pooling layer. Moreover, N \geq 0 \ (and usually N \leq 3), M \geq 0, K \geq 0 \ (and usually K < 3). For example, here are some common ConvNet architectures you may see that follow this pattern:

- \text{INPUT} \rightarrow \text{FC}, implements a linear classifier. Here N = M = K = 0.
- \text{INPUT} \rightarrow \text{CONV} \rightarrow \text{RELU} \rightarrow \text{FC}
- \text{INPUT} \rightarrow [\text{CONV} \rightarrow \text{RELU} \rightarrow \text{POOL}]^2 \rightarrow \text{FC} \rightarrow \text{RELU} \rightarrow \text{FC}. Here we see that there is a single CONV layer between every POOL layer.
- \text{INPUT} \rightarrow [\text{CONV} \rightarrow \text{RELU} \rightarrow \text{CONV} \rightarrow \text{RELU} \rightarrow \text{POOL}]^3 \rightarrow [\text{FC} \rightarrow \text{RELU}]^2 \rightarrow \text{FC}. Here we see two CONV layers stacked before every POOL layer. This is generally a good idea for larger and deeper networks, because multiple stacked CONV layers can develop more complex features of the input volume before the destructive pooling operation.
Taking care of downsampling

• At some point(s) in the network, we need to reduce the size
  • If conv layers do not downsize, then only pooling layers take care of downsampling
  • If conv layers also downsize, you need to be careful about strides etc. so that
    (i) the dimension requirements of all layers are satisfied and
    (ii) all layers tile up properly.
• S=1 seems to work well in practice
• However, for bigger input volumes, you may try bigger strides
Trade-offs in architecture

• Between filter size and number of layers (depth)
  • Keep the layer widths fixed.
  • “When the time complexity is roughly the same, the deeper networks with smaller filters show better results than the shallower networks with larger filters.”

• Between layer width and number of layers (depth)
  • Keep the size of the filters fixed.
  • “We find that increasing the depth leads to considerable gains, even the width needs to be properly reduced.”

• Between filter size and layer width
  • Keep the number of layers (depth) fixed.
  • No significant difference

4.4. Is Deeper Always Better?

The above results have shown the priority of depth for improving accuracy. With the above trade-offs, we can have a much deeper model if we further decrease width/filter sizes and increase depth. However, in experiments we find that the accuracy is stagnant or even reduced in some of our very deep attempts. There are two possible explanations: (1) the width/filter sizes are reduced overly and may harm the accuracy, or (2) overly increasing the depth will degrade the accuracy even if the other factors are not traded. To understand the main reason, in this subsection we do not constrain the time complexity but solely increase the depth without other changes.
Memory

Main sources of memory load:

• Activation maps:
  • Training: They need to be kept during training so that backpropagation can be performed
  • Testing: No need to keep the activations of earlier layers

• Parameters:
  • The weights, their gradients and also another copy if momentum is used

• Data:
  • The originals + their augmentations

• If all these don’t fit into memory,
  • Load your data batch by batch from disk
  • Decrease the size of your batches
Memory constraints

- Using smaller RFs with more layers means more memory since you need to store more activation maps.

- In such memory-scarce cases,
  - the first layer may use bigger RFs with $S > 1$
  - information loss from the input volume may be less critical than the following layers.

- E.g., AlexNet uses RFs of 11x11 and $S = 4$ for the first layer.
Finetuning

1. If the new dataset is small and similar to the original dataset used to train the CNN:
   • Finetuning the whole network may lead to overfitting
     • Just train the newly added layer

2. If the new dataset is big and similar to the original dataset:
   • The more, the merrier: go ahead and train the whole network

3. If the new dataset is small and different from the original dataset:
   • Not a good idea to train the whole network
   • However, add your new layer not to the top of the network, since those parts are very dataset (problem) specific
     • Add your layer to earlier parts of the network

4. If the new dataset is big and different from the original dataset:
   • We can “finetune” the whole network
   • This amounts to a new training problem by initializing the weights with those of another network
Many different mechanisms

• Visualize layer activations
• Visualize the weights (i.e., filters)
• Visualize examples that maximally activate a neuron
• Visualize a 2D embedding of the inputs based on their CNN codes
• Occlude parts of the window and see how the prediction is affected
• Data gradients
Embed the codes in a lower-dimensional space

• Place images into a 2D space such that images which produce similar CNN codes are placed close.
• You can use, e.g., t-Distributed Stochastic Neighbor Embedding (t-SNE)

Figure 1: Illustration of t-SNE on MNIST dataset

Figure: Laurens van der Maaten and Geoffrey Hinton
Data gradients

- The gradient with respect to the input is high for pixels which are on the object.

\[
S_c(I) = w_c^T I + b_c, \tag{2}
\]

where the image \( I \) is represented in the vectorised (one-dimensional) form, and \( w_c \) and \( b_c \) are respectively the weight vector and the bias of the model. In this case, it is easy to see that the magnitude of elements of \( w \) defines the importance of the corresponding pixels of \( I \) for the class \( c \).

In the case of deep ConvNets, the class score \( S_c(I) \) is a highly non-linear function of \( I \), so the reasoning of the previous paragraph can not be immediately applied. However, given an image \( I_0 \), we can approximate \( S_c(I) \) with a linear function in the neighbourhood of \( I_0 \) by computing the first-order Taylor expansion:

\[
S_c(I) \approx w^T I + b, \tag{3}
\]

where \( w \) is the derivative of \( S_c \) with respect to the image \( I \) at the point (image) \( I_0 \):

\[
w = \frac{\partial S_c}{\partial I}
\bigg|_{I_0}. \tag{4}
\]
Class Activation Maps

Weighted combination of the feature maps before GAP:

\[ M(x, y) = \sum_{k} w_k^c f_k(x, y) \]

Feature inversion

- Learns to reconstruct an image from its representation

This section introduces our method to compute an approximate inverse of an image representation. This is formulated as the problem of finding an image whose representation best matches the one given \[ \Phi : \mathbb{R}^{H \times W \times C} \rightarrow \mathbb{R}^d \]. Formally, given a representation function \( \Phi : \mathbb{R}^{H \times W \times C} \rightarrow \mathbb{R}^d \) and a representation \( \Phi_0 = \Phi(x_0) \) to be inverted, reconstruction finds the image \( x \in \mathbb{R}^{H \times W \times C} \) that minimizes the objective:

\[
x^* = \arg\min_{x \in \mathbb{R}^{H \times W \times C}} \ell(\Phi(x), \Phi_0) + \lambda \mathcal{R}(x)
\]

where the loss \( \ell \) compares the image representation \( \Phi(x) \) to the target one \( \Phi_0 \) and \( \mathcal{R} : \mathbb{R}^{H \times W \times C} \rightarrow \mathbb{R} \) is a regulariser capturing a natural image prior.

- Regularization term here is the key factor, e.g. a combination of the two terms:

\[
\mathcal{R}_\alpha(x) = \|x\|_\alpha^\alpha, \quad \mathcal{R}_{\mathcal{V}, \beta}(x) = \sum_{i,j} \left( (x_{i,j+1} - x_{ij})^2 + (x_{i+1,j} - x_{ij})^2 \right)^{\beta/2}
\]
Feature inversion with perceptual losses

Figure from Johnson, Alahi, and Fei-Fei, “Perceptual Losses for Real-Time Style Transfer and Super-Resolution”, ECCV 2016.
Fooling ConvNets

• Given an image $I$ labeled as $l_1$, find minimum “$r$” (noise) such that $I + r$ is classified as a different label, $l_2$.

• I.e., minimize:

$$\arg\min_{r} \text{loss}(I + r, l_2) + c|r|$$
LeNet (1998)

• For reading zip codes and digits

Euclidean RBF:

\[ y_i = \sum_j (x_j - w_{ij})^2. \]
AlexNet (2012)

- Popularized CNN in computer vision & pattern recognition
  - ImageNet ILSVRC challenge 2012 winner
- Similar to LeNet
  - Deeper & bigger
  - Many CONV layers on top of each other (rather than adding immediately a pooling layer after a CONV layer)
  - Uses GPU
- 650K neurons. 60M parameters. Trained on 2 GPUs for a week.
GoogleNet (2014)

- ImageNet 2014 winner
- Contributions:
  - Inception module
    - Dramatically reduced parameters (from 60M in AlexNet to 4M)
  - Avg Pooling at the top, instead of fully-connected layer ➔ Reduced number of parameters
- Motivation:
  - Going bigger (in depth or width) means too many parameters.
  - Go bigger by maintaining sparse connections.
Inception module: “network in network”  
(inspired from Lin et al., 2013)

- Concatenation is performed along the “columns” (depth).
  - The output of inception layers must have the same size.
- The naïve version has a tendency to blow up in number of channels.
  - Why? Max-pooling does not change the number of channels. When concatenated with other filter responses, number of channels increase with every layer.
  - Solution: Do 1x1 convolution to decrease the number of channels.
    - Also called “bottleneck”.
- In order to decrease the computational complexity of 3x3 and 5x5 pooling, they are also preceded by 1x1 convolution (i.e., the number of channels are reduced).
VGGNet (2014)

- ImageNet runner up in 2014
- Contribution:
  - Use small RFs & increase depth as much as possible
  - 16 CONV/FC layers.
  - 3x3 CONVs and 2x2 pooling from beginning to the end
- Although performs slightly worse than GoogleNet in image classification, VGGNet may perform better at other tasks (such as transfer learning problems).
- Downside: Needs a lot of memory & parameters (140M)

- Increasing the depth naively may not give you better performance after a number of depths
  - Why?
    - This is shown to be not due to overfitting (since training error also gets worse) or vanishing gradients (suitable non-linearities used)
    - Accuracy is somehow saturated. Not clear why. Though reported in several studies.
  - Solution: Make shortcut connections

- Residual (shortcut) connections

\[
\mathcal{F}(x) = \text{weight layer} \rightarrow \text{relu} \rightarrow \text{weight layer} \rightarrow \text{identity} \rightarrow \mathcal{F}(x) + x \rightarrow \text{relu}
\]

Figure 2. Residual learning: a building block.

Figure 5. A deeper residual function $\mathcal{F}$ for ImageNet. Left: a building block (on $56 \times 56$ feature maps) as in Fig. 3 for ResNet-34. Right: a “bottleneck” building block for ResNet-50/101/152.
Effect of residual connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The vertical axis is logarithmic to show dynamic range. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

VISUALIZING THE LOSS LANDSCAPE OF NEURAL NETS

Hao Li¹, Zheng Xu², Gavin Taylor², Christoph Studer³, Tom Goldstein¹
¹University of Maryland, College Park, ²United States Naval Academy, ³Cornell University
{haoli,zxu,ctor}@cs.umd.edu, taylor@usna.edu, studer@cornell.edu
Figure 1. **Left:** A block of ResNet [14]. **Right:** A block of ResNeXt with cardinality = 32, with roughly the same complexity. A layer is shown as (# in channels, filter size, # out channels).
Today

• Convolutional Neural Networks (CNNs)
  • CNN applications

• Recurrent Neural Networks (RNNs)

<table>
<thead>
<tr>
<th>Week</th>
<th>Topics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 12 (27 May)</td>
<td>CNNs, RNNs</td>
</tr>
<tr>
<td>Week 13 (3 June)</td>
<td>RNNs</td>
</tr>
<tr>
<td>Week 14 (10 June)</td>
<td>RNNs + Attention</td>
</tr>
<tr>
<td>Week 15 (17 June)</td>
<td>Recent Studies/Approaches</td>
</tr>
</tbody>
</table>
# Administrative Issues

- **Programming Assignment 2 (PA2):**
  - Deadline: 5 June
- **Programming Assignment 3 (PA3):**
  - Tentative Deadline: 19 June
- **Final exam:**
  - Tentative Deadline: 23-26 June.
- **Projects:**
  - Tentative Deadline: 5 July.

---

**May**

<table>
<thead>
<tr>
<th>S</th>
<th>M</th>
<th>T</th>
<th>W</th>
<th>T</th>
<th>F</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td>29</td>
<td>30</td>
<td>31</td>
<td><strong>PA2</strong></td>
<td><strong>PA3</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**June**

<table>
<thead>
<tr>
<th>S</th>
<th>M</th>
<th>T</th>
<th>W</th>
<th>T</th>
<th>F</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td>29</td>
<td>30</td>
<td>31</td>
<td><strong>PA2</strong></td>
<td><strong>PA3</strong></td>
<td><strong>Final</strong></td>
<td></td>
</tr>
</tbody>
</table>

**July**

<table>
<thead>
<tr>
<th>S</th>
<th>M</th>
<th>T</th>
<th>W</th>
<th>T</th>
<th>F</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td><strong>Project</strong></td>
<td><strong>Final</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**August**

<table>
<thead>
<tr>
<th>S</th>
<th>M</th>
<th>T</th>
<th>W</th>
<th>T</th>
<th>F</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

17 June : Last day of classes  
09-12 July : Bayram  
12 July : Submitting grades  
17 July : Incomplete grades
Object Detection with CNNs
Visual Detection

(a) Object Detection  (b) Keypoint Detection
(c) Instance Segmentation  (d) Panoptic Segmentation

Images are from the COCO dataset.
Object detection

Image: https://analyticsprofile.com/machine-learning/object-detection-basic-tutorial-in-python/
Object detection

Regions of Interest (RoIs): Regions that are likely to contain objects
Object detection

\[ f(\cdot) \]

Classification

Object category (car, person, ..)

Localization

Object position & size \((x, y, w, h)\)

\[ (x, y) \]

\[ h \]

\[ w \]
Object detection

$\mathcal{L} = \mathcal{L}_c + w_r \mathcal{L}_r$

Classification Loss $\mathcal{L}_c$

Localisation Loss $\mathcal{L}_r$

Object category (car, person, ..)

Object position & size (x, y, w, h)

Regions with CNN (R-CNN)

- A very straightforward application
- Start from a pre-trained model (trained on ImageNet)
- Finetune using the new data available

Fig. 1. Object detection system overview. Our system (1) takes an input image, (2) extracts around 2000 bottom-up region proposals, (3) computes features for each proposal using a large convolutional network (CNN), and then (4) classifies each region using class-specific linear SVMs. We trained an R-CNN that achieves a mean average precision (mAP) of 62.9% on PASCAL VOC 2010. For comparison, [21] reports 35.1% mAP using the same region proposals, but with a spatial pyramid and bag-of-visual-words approach. The popular deformable part models perform at 33.4%. On the 200-class ILSVRC2013 detection dataset, we trained an R-CNN with a mAP of 31.4%, a large improvement over OverFeat [19], which had the previous best result at 24.3% mAP.
Fast R-CNN

• Feature Extraction: From the whole image once using a pre-trained network
• RoI Projection: Map coordinates of a box to the space of CNN features
• RoI Pooling: Max-pool features to have a fixed-size feature vector (e.g. 7x7)
• Use FC layers for cls & reg

Figure 1. Fast R-CNN architecture. An input image and multiple regions of interest (RoIs) are input into a fully convolutional network. Each RoI is pooled into a fixed-size feature map and then mapped to a feature vector by fully connected layers (FCs). The network has two output vectors per RoI: softmax probabilities and per-class bounding-box regression offsets. The architecture is trained end-to-end with a multi-task loss.
RoI Pooling

- Goal: obtain fixed-size features from regions of different sizes

Figure: https://deepsense.ai/region-of-interest-pooling-explained/
Faster R-CNN

- Replaced region-proposal method with a network
- Followed by the “Fast R-CNN” classifier
- The two networks share some convolutional layers

Figure 2: Faster R-CNN is a single, unified network for object detection. The RPN module serves as the ‘attention’ of this unified network.
Mask R-CNN

Figure 1. The Mask R-CNN framework for instance segmentation.

Figure 2. Mask R-CNN results on the COCO test set. These results are running at 5 fps. Masks are shown in color, and bounding box, category...
You Look Only Once (YOLO)

Single-stage method:
- Perform classification & localization for a fixed set of potential regions.
- Very fast compared to other detectors

Each cell predicts:
- Coordinates \((x, y, w, h)\) & confidence for B boxes
- Confidence: \(\Pr(Object) \times \text{IoU}^{\text{truth}}_{\text{pred}}\)
- \(C\) classes: \(\Pr(Class_i | Object)\)

Test time:
- \(\Pr(Class_i | Object) \times \Pr(Object) \times \text{IoU}^{\text{truth}}_{\text{pred}} = \Pr(Class_i) \times \text{IoU}^{\text{truth}}_{\text{pred}}\)

For PASCAL VOC: \(S = 7, B = 2, C = 20\)
Single-Shot Detector (SSD)

- Allows more bounding boxes per location
- Multi-scale detection
- 8372 BBs per class per image
SSD: Single Shot MultiBox Detector

Wei Liu¹, Dragomir Anguelov², Dumitru Erhan³, Christian Szegedy³, Scott Reed³, Cheng-Yang Fu¹, Alexander C. Berg⁴

(2016)

Single-Shot Detector (SSD)
Feature Pyramid Networks for Object Detection

- Used for feature extraction
- Can be used for region proposal, object detection or segmentation.

Figure 4. FPN for object segment proposals. The feature pyramid is constructed with identical structure as for object detection. We
Focal Loss for Dense Object Detection (RetinaNet)

Figure 3. The one-stage RetinaNet network architecture uses a Feature Pyramid Network (FPN) [20] backbone on top of a feedforward ResNet architecture [16] (a) to generate a rich, multi-scale convolutional feature pyramid (b). To this backbone RetinaNet attaches two subnetworks, one for classifying anchor boxes (c) and one for regressing from anchor boxes to ground-truth object boxes (d). The network design is intentionally simple, which enables this work to focus on a novel focal loss function that eliminates the accuracy gap between our one-stage detector and state-of-the-art two-stage detectors like Faster R-CNN with FPN [20] while running at faster speeds.

The highest accuracy object detectors to date are based on a two-stage approach popularized by R-CNN, where a classifier is applied to a sparse set of candidate object locations. In contrast, one-stage detectors that are applied over a regular, dense sampling of possible object locations have the potential to be faster and simpler, but have trailed the accuracy of two-stage detectors thus far. In this paper, we investigate why this is the case. We discover that the extreme foreground-background class imbalance encountered during training of dense detectors is the central cause. We propose to address this class imbalance by reshaping the standard cross entropy loss such that it down-weights the loss assigned to well-classified examples. Our novel Focal Loss focuses training on a sparse set of hard examples and prevents the vast number of easy negatives from overwhelming the detector during training. To evaluate the effectiveness of our loss, we design and train a simple dense detector we call RetinaNet. Our results show that when trained with the focal loss, RetinaNet is able to match the speed of previous one-stage detectors while surpassing the accuracy of all existing state-of-the-art two-stage detectors. Code is at: https://github.com/facebookresearch/Detector.
RetinaNet: Focal Loss

from the cross entropy (CE) loss for binary classification:\(^1\):

\[
CE(p, y) = \begin{cases} 
- \log(p) & \text{if } y = 1 \\
- \log(1 - p) & \text{otherwise.}
\end{cases}
\]  

(1)

In the above \( y \in \{\pm 1\} \) specifies the ground-truth class and \( p \in [0, 1] \) is the model’s estimated probability for the class with label \( y = 1 \). For notational convenience, we define \( p_t \):

\[
p_t = \begin{cases} 
p & \text{if } y = 1 \\
1 - p & \text{otherwise},
\end{cases}
\]  

(2)

and rewrite \( CE(p, y) = CE(p_t) = - \log(p_t) \).

More formally, we propose to add a modulating factor \((1 - p_t)\gamma\) to the cross entropy loss, with tunable focusing parameter \(\gamma \geq 0\). We define the focal loss as:

\[
FL(p_t) = -(1 - p_t)^\gamma \log(p_t).
\]  

(4)

Figure 1. We propose a novel loss we term the Focal Loss that adds a factor \((1 - p_t)^\gamma\) to the standard cross entropy criterion. Setting \(\gamma > 0\) reduces the relative loss for well-classified examples \((p_t > .5)\), putting more focus on hard, misclassified examples. As our experiments will demonstrate, the proposed focal loss enables training highly accurate dense object detectors in the presence of vast numbers of easy background examples.
FCOS: Fully Convolutional One-stage Object Detection

• Create a segmentation map from object centers
• Treat object detection as segmentation of object centers
• For positive samples, regress box coordinates

$$L(\{p_{x,y}\}, \{t_{x,y}\}) = \frac{1}{N_{pos}} \sum_{x,y} L_{cls}(p_{x,y}, c^{*}_{x,y})$$
$$+ \frac{\lambda}{N_{pos}} \sum_{x,y} \mathbb{I}_{\{c^{*}_{x,y} > 0\}} L_{reg}(t_{x,y}, t^{*}_{x,y}).$$
Some of our work


Other Applications
Takes 19 layer VGG as the base (no FC layers)

Max pooling is replaced by avg pooling since it produced more appealing results
Figure 1: Convolutional Neural Network (CNN). A given input image is represented as a set of filtered images at each processing stage in the CNN. While the number of different filters increases along the processing hierarchy, the size of the filtered images is reduced by some downsampling mechanism (e.g., max-pooling) leading to a decrease in the total number of units per layer of the network. **Content Reconstructions.** We can visualise the information at different processing stages in the CNN by reconstructing the input image from only knowing the network’s responses in a particular layer. We reconstruct the input image from four layers: `conv1_1` (a), `conv2_1` (b), `conv3_1` (c), `conv4_1` (d), and `conv5_1` (e) of the original VGG-Net. We find that reconstruction from lower layers is almost perfect (a,b,c). In higher layers of the network, detailed pixel information is lost while the high-level content of the image is preserved (d,e). **Style Reconstructions.** On top of the original CNN representations we built a new feature space that captures the style of an input image. The style representation computes correlations between the different features in different layers of the CNN. We reconstruct the style of the input image from style representations built on different subsets of CNN layers (`conv1_1`, `conv2_1`, `conv3_1`, `conv4_1`, and `conv5_1`). This creates images that match the style of a given image on an increasing scale while discarding information of the global arrangement of the scene.
Content reconstruction: gradient descent on a white-noise image to find an image that matches the filter responses.

\[ L_{\text{content}}(\bar{p}, \bar{x}, l) = \frac{1}{2} \sum_{i,j} (F_{ij}^l - P_{ij}^l)^2. \]  

Style representation:

Gram matrix \( G^l \in \mathcal{R}^{N_t \times N_t} \), where \( G_{ij}^l \) is the inner product between the vectorised feature map \( i \) and \( j \) in layer \( l \):

\[ G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l. \]
Style reconstruction:

To generate a texture that matches the style of a given image (Fig 1, style reconstructions), we use gradient descent from a white noise image to find another image that matches the style representation of the original image. This is done by minimising the mean-squared distance between the entries of the Gram matrix from the original image and the Gram matrix of the image to be generated. So let $\tilde{a}$ and $\tilde{x}$ be the original image and the image that is generated and $A_l$ and $G_l$ their respective style representations in layer $l$. The contribution of that layer to the total loss is then

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G_{ij}^l - A_{ij}^l)^2$$

(4)

and the total loss is

$$\mathcal{L}_{\text{style}}(\tilde{a}, \tilde{x}) = \sum_{l=0}^{L} w_l E_l$$

(5)

where $w_l$ are weighting factors of the contribution of each layer to the total loss (see below for specific values of $w_l$ in our results).
Overall loss:

\[ L_{\text{total}}(\vec{p}, \vec{a}, \vec{x}) = \alpha L_{\text{content}}(\vec{p}, \vec{x}) + \beta L_{\text{style}}(\vec{a}, \vec{x}) \]
Figure 3: Detailed results for the style of the painting *Composition VII* by Wassily Kandinsky. The rows show the result of matching the style representation of increasing subsets of the CNN layers (see Methods). We find that the local image structures captured by the style representation increase in size and complexity when including style features from higher layers of the network. This can be explained by the increasing receptive field sizes and feature complexity along the network’s processing hierarchy. The columns show different relative weightings between the content and style reconstruction. The number above each column indicates the ratio $\alpha/\beta$ between the emphasis on matching the content of the photograph and the style of the artwork (see Methods).
Other Applications

• Tracking (Bazzani et. al. 2010, and many others)

• Pose estimation (Toshev et al. 2013, Jain et al., 2013, ...)

• Caption generation (Vinyals et al. 2015, Xu et al. 2015, ...)

A woman is throwing a frisbee in a park.  A dog is standing on a hardwood floor.  A stop sign is on a road with a mountain in the background.
Convoultional networks for music recommendation

Image from: http://benanne.github.io/2014/08/05/spotify-cnns.html
To wrap up
CNNs: summary & future directions

• Less parameters
• Allows going deeper
• High flexibility
  • In operations
  • In organization of layers
  • In the overall architecture etc.
• Future directions:
  • Understanding them better
  • Making them deeper, faster and more efficient
  • Compressing a big network into a smaller & cheaper one.
  • ...

Spring 2022
Sinan Kalkan
Sequence Labeling/Modeling: Motivation
Why do we need them?

A. Graves, “Supervised Sequence Labelling with Recurrent Neural Networks”, 2012.
Different types of sequence learning / recognition problems

• Sequence Classification
  • A sequence to a label
  • E.g., recognizing a single spoken word
  • Length of the sequence is fixed
  • Why RNNs then? Because sequential modeling provides robustness against translations and distortions.

• Segment Classification
  • Segments in a sequence correspond to labels

• Temporal Classification
  • General case: sequence (input) to sequence (label) modeling.
  • No clue about where input or label starts.
Recurrent Neural Networks
Recurrent Neural Networks (RNNs)

- RNNs are very powerful because:
  - Distributed hidden state that allows them to store a lot of information about the past efficiently.
  - Non-linear dynamics that allows them to update their hidden state in complicated ways.
- With enough neurons and time, RNNs can compute anything that can be computed by your computer.
- More formally, RNNs are Turing complete.
Some examples

- Temporal pattern recognition

  - OUTPUT
  - CONTEXT
  - INPUT

  e.g., speech recognition
  e.g., event recognition
  e.g., natural language understand

- Sequence generation

  - OUTPUT
  - PLAN
  - CONTEXT

  e.g., speech production
  e.g., motor control
  e.g., planning and acting

- Pattern completion / constraint satisfaction

  - GANG
  - AGE
  - EDUCATION
  - INSTANCE
  - OCCUPATION
  - NAME
Some examples

Jordan Networks

Elman Networks

“context” neurons

Figs: David Kriesel
Challenge

• Back propagation is designed for feedforward nets
• What would it mean to back propagate through a recurrent network?
  • error signal would have to travel back in time
Unfolding

Feed-forward networks

Recurrent networks
Unfolding (another example)

Figure: Michael Mozer
How an RNN works

Alec Radford
You can stack them too
Unfolding implications

• Entails duplication of weights => weight sharing
• Sharing weights means their gradients will be accumulated over time and reflected on the weights
• Unfolded network has the same dynamics of the RNN for a fixed number of time steps!
Back-propagation Through Time
Feedforward through Vanilla RNN

\( \mathbf{h}_1 = \tanh (W^{xh} \cdot \mathbf{x}_1 + W^{hh} \cdot \mathbf{h}_0) \)

\( \hat{\mathbf{y}}_1 = \text{softmax} (W^{hy} \cdot \mathbf{h}_1) \)

\( L_1 = CE(\hat{\mathbf{y}}_1, \mathbf{y}_1) \)
The Vanilla RNN Model

First time-step \((t = 1)\):

\[
\mathbf{h}_1 = \tanh(W^{xh} \cdot \mathbf{x}_1 + W^{hh} \cdot \mathbf{h}_0) \\
\hat{\mathbf{y}}_1 = \text{softmax}(W^{hy} \cdot \mathbf{h}_1) \\
L_1 = CE(\hat{\mathbf{y}}_1, \mathbf{y}_1)
\]

In general:

\[
\mathbf{h}_t = \tanh(W^{xh} \cdot \mathbf{x}_t + W^{hh} \cdot \mathbf{h}_{t-1}) \\
\hat{\mathbf{y}}_t = \text{softmax}(W^{hy} \cdot \mathbf{h}_t) \\
L_t = CE(\hat{\mathbf{y}}_t, \mathbf{y}_t)
\]

In total:

\[
L = \sum_t L_t
\]
Backpropagation through Vanilla RNN

\[
\frac{\partial L}{\partial W^{hy}} = \?
\]

\[
= \frac{\partial L}{\partial \hat{y}_n} \frac{\partial \hat{y}_n}{\partial W^{hy}} + \frac{\partial L}{\partial \hat{y}_{n-1}} \frac{\partial \hat{y}_{n-1}}{\partial W^{hy}} + \cdots + \frac{\partial L}{\partial \hat{y}_1} \frac{\partial \hat{y}_1}{\partial W^{hy}}
\]

\[
= \sum_{t=1..n} \frac{\partial L}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial W^{hy}}
\]

In general:

\[
\begin{align*}
  h_t &= \tanh(W^{xh} \cdot x_t + W^{hh} \cdot h_{t-1}) \\
  \hat{y}_t &= \text{softmax}(W^{hy} \cdot h_t) \\
  L_t &= \text{CE}(\hat{y}_t, y_t)
\end{align*}
\]

In total:

\[
L = \sum_t L_t
\]
Backpropagation through Vanilla RNN

\[
\frac{\partial L}{\partial W^{hh}} = ?
\]

\[
= \frac{\partial L}{\partial h_n} \frac{\partial h_n}{\partial W^{hh}} + \frac{\partial L}{\partial h_{n-1}} \frac{\partial h_{n-1}}{\partial W^{hh}} + \cdots + \frac{\partial L}{\partial h_1} \frac{\partial h_1}{\partial W^{hh}}
\]

\[
\frac{\partial L}{\partial h_t} = \frac{\partial L}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} + \frac{\partial L}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_t}
\]

In general:
\[
h_t = \tanh(W^{xh} \cdot x_t + W^{hh} \cdot h_{t-1})
\]
\[
\hat{y}_t = \text{softmax}(W^{hy} \cdot h_t)
\]
\[
L_t = CE(\hat{y}_t, y_t)
\]

In total:
\[
L = \sum_t L_t
\]
\[
\frac{\partial C_t}{\partial W} = \sum_{t'=1}^{t} \frac{\partial C_t}{\partial h_t} \frac{\partial h_t}{\partial h_{t'}} \frac{\partial h_{t'}}{\partial W}, \text{ where } \frac{\partial h_t}{\partial h_{t'}} = \prod_{k=t'+1}^{t} \frac{\partial h_k}{\partial h_{k-1}}
\]
Backpropagation through Vanilla RNN

$$\frac{\partial L}{\partial W^{xh}} = ?$$

$$= \frac{\partial L}{\partial \mathbf{h}_n} \frac{\partial \mathbf{h}_n}{\partial W^{xh}} + \frac{\partial L}{\partial \mathbf{h}_{n-1}} \frac{\partial \mathbf{h}_{n-1}}{\partial W^{xh}} + \cdots + \frac{\partial L}{\partial \mathbf{h}_1} \frac{\partial \mathbf{h}_1}{\partial W^{xh}}$$

$$\frac{\partial L}{\partial \mathbf{h}_t} = \frac{\partial L}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial \mathbf{h}_t} + \frac{\partial L}{\partial \mathbf{h}_{t+1}} \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}$$

(calculated before)

In general:

$$\mathbf{h}_t = \tanh(W^{xh} \cdot \mathbf{x}_t + W^{hh} \cdot \mathbf{h}_{t-1})$$

$$\hat{\mathbf{y}}_t = \text{softmax}(W^{hy} \cdot \mathbf{h}_t)$$

$$L_t = CE(\hat{\mathbf{y}}_t, \mathbf{y}_t)$$

In total:

$$L = \sum_t L_t$$
An irritating extra issue

- We need to specify the initial activity state of all the hidden and output units.
- We could just fix these initial states to have some default value like 0.5.
- But it is better to treat the initial states as learned parameters.
- We learn them in the same way as we learn the weights.
  - Start off with an initial random guess for the initial states.
  - At the end of each training sequence, backpropagate through time all the way to the initial states to get the gradient of the error function with respect to each initial state.
  - Adjust the initial states by following the negative gradient.

Slide: Hinton
Initializing parameters

• Since an unfolded RNN is a deep MLP, we can use Xavier initialization.
The problem of exploding or vanishing gradients

• What happens to the magnitude of the gradients as we backpropagate through many layers?
  – If the weights are small, the gradients shrink exponentially.
  – If the weights are big the gradients grow exponentially.
• Typical feed-forward neural nets can cope with these exponential effects because they only have a few hidden layers.

• In an RNN trained on long sequences (e.g. 100 time steps) the gradients can easily explode or vanish.
  – We can avoid this by initializing the weights very carefully.
• Even with good initial weights, it’s very hard to detect that the current target output depends on an input from many time-steps ago.
  – So RNNs have difficulty dealing with long-range dependencies.

Slide: Hinton
Exploding and vanishing gradients problem

• **Solution 1**: Gradient clipping for exploding gradients:

  ![Algorithm 1 Pseudo-code for norm clipping](image)

  • For vanishing gradients: Regularization term that penalizes changes in the magnitudes of back-propagated gradients

\[
\Omega = \sum_k \Omega_k = \sum_k \left( \left\| \frac{\partial \mathcal{E}}{\partial x_{k+1}} \frac{\partial x_{k+1}}{\partial x_k} \right\| - 1 \right)^2
\]
Exploding and vanishing gradients problem

• Solution 2:
  • Use methods like LSTM
Abstract

Learning to store information over extended time intervals via recurrent backpropagation takes a very long time, mostly due to insufficient, decaying error backflow. We briefly review Hochreiter’s 1991 analysis of this problem, then address it by introducing a novel, efficient, gradient-based method called “Long Short-Term Memory” (LSTM). Truncating the gradient where this does not do harm, LSTM can learn to bridge minimal time lags in excess of 1000 discrete time steps by enforcing constant error flow through “constant error carrousels” within special units. Multiplicative gate units learn to open and close access to the constant error flow. LSTM is local in space and time; its computational complexity per time step and weight is O(1). Our experiments with artificial data involve local, distributed, real-valued, and noisy pattern representations. In comparisons with RTRL, BPTT, Recurrent Cascade-Correlation, Elman nets, and Neural Sequence Chunking, LSTM leads to many more successful runs and learns much faster. LSTM also solves complex, artificial long time lag tasks that have never been solved by previous recurrent network algorithms.

Long Short Term Memory (LSTM)
RNN

• Basic block diagram

(C) Dhruv Batra
Image Credit: Christopher Olah (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)
Key Problem

- Learning long-term dependencies is hard

(C) Dhruv Batra
Image Credit: Christopher Olah (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)
Long Short Term Memory (LSTM)

- Hochreiter & Schmidhuber (1997) solved the problem of getting an RNN to remember things for a long time (like hundreds of time steps).
- They designed a memory cell using logistic and linear units with multiplicative interactions.
- Information gets into the cell whenever its “write” gate is on.
- The information stays in the cell so long as its “keep” gate is on.
- Information can be read from the cell by turning on its “read” gate.
Meet LSTMs

• How about we explicitly encode memory?
LSTM in detail

• We first compute an activation vector, \( a \):
  \[ a = W_x x_t + W_h h_{t-1} + b \]

• Split this into four vectors of the same size:
  \[ a_i, a_f, a_o, a_g \leftarrow a \]

• We then compute the values of the gates:
  \[ i = \sigma(a_i) \quad f = \sigma(a_f) \quad o = \sigma(a_o) \quad g = \tanh(a_g) \]
  where \( \sigma \) is the sigmoid.

• The next cell state \( c_t \) and the hidden state \( h_t \):
  \[ c_t = f \odot c_{t-1} + i \odot g \]
  \[ h_t = o \odot \tanh(c_t) \]
  where \( \odot \) is the element-wise product of vectors

Alternative formulation:

\[ i_t = g(W_{zi} x_t + W_{hi} h_{t-1} + b_i) \]
\[ f_t = g(W_{zf} x_t + W_{hf} h_{t-1} + b_f) \]
\[ o_t = g(W_{zo} x_t + W_{ho} h_{t-1} + b_o) \]

Eqs: Karpathy
LSTMs Intuition: Memory

• Cell State / Memory

Image Credit: Christopher Olah (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)

(C) Dhruv Batra
LSTMs Intuition: Forget Gate

• Should we continue to remember this “bit” of information or not?

\[ f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \]
LSTMs Intuition: Input Gate

• Should we update this “bit” of information or not?
  • If so, with what?

\[ i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \]
\[ \tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \]
LSTMs Intuition: Memory Update

- Forget that + memorize this

\[ C_t = f_t \times C_{t-1} + i_t \times \tilde{C}_t \]
LSTMs Intuition: Output Gate

• Should we output this “bit” of information to “deeper” layers?

\[ o_t = \sigma \left( W_o \left[ h_{t-1}, x_t \right] + b_o \right) \]
\[ h_t = o_t \times \tanh \left( C_t \right) \]
LSTMs

• A pretty sophisticated cell
LSTM Variants #1: Peephole Connections

• Let gates see the cell state / memory

\[
\begin{align*}
    f_t &= \sigma (W_f \cdot [C_{t-1}, h_{t-1}, x_t] + b_f) \\
    i_t &= \sigma (W_i \cdot [C_{t-1}, h_{t-1}, x_t] + b_i) \\
    o_t &= \sigma (W_o \cdot [C_t, h_{t-1}, x_t] + b_o)
\end{align*}
\]
LSTM Variants #2: Coupled Gates

• Only memorize new if forgetting old

\[ C_t = f_t \times C_{t-1} + (1 - f_t) \times \tilde{C}_t \]
LSTM Variants #3: Gated Recurrent Units

• Changes:
  • No explicit memory; memory = hidden output
  • $Z = \text{memorize new and forget old}$

\[
\begin{align*}
  z_t &= \sigma (W_z \cdot [h_{t-1}, x_t]) \\
  r_t &= \sigma (W_r \cdot [h_{t-1}, x_t]) \\
  \tilde{h}_t &= \tanh (W \cdot [r_t \ast h_{t-1}, x_t]) \\
  h_t &= (1 - z_t) \ast h_{t-1} + z_t \ast \tilde{h}_t
\end{align*}
\]
LSTM vs. GRU

Figure 1: Activations — c for LSTM and h for GRU — for networks trained on $a^n b^n$ and $a^n b^n c^n$. The LSTM has clearly learned to use an explicit counting mechanism, in contrast with the GRU.
ConvLSTM

Figure 2: Inner structure of ConvLSTM

https://medium.com/neuronio/an-introduction-to-convlstm-55c9025563a7
A bio-inspired bistable recurrent cell allows for long-lasting memory

Nicolas Vecoven  
University of Liège  
nvecoven@uliege.be

Damien Ernst  
University of Liège

Guillaume Drion  
University of Liège

We have taken inspiration from biological neuron bistability to embed Recurrent Neural Nets (RNN) with long-lasting memory at the cellular level.

Excellent performances on time-series which require very long memory.


#DeepLearning #ArtificialIntelligence
Reference

• A very detailed explanation with nice figures

http://colah.github.io/posts/2015-08-Understanding-LSTMs/
Figure 1. Architectural elements in a TCN. (a) A dilated causal convolution with dilation factors $d = 1, 2, 4$ and filter size $k = 3$. The receptive field is able to cover all values from the input sequence. (b) TCN residual block. An 1x1 convolution is added when residual input and output have different dimensions. (c) An example of residual connection in a TCN. The blue lines are filters in the residual function, and the green lines are identity mappings.