Paper 8: Image Searching and Modelling Using Machine Learning

Handout 2: Convolutional Neural Networks

Roberto Cipolla and Matthew Johnson
May 2016
Recognising Digits

An important task in the history of artificial neural networks is the recognition of handwritten digits, driven by a dataset known as MNIST.

The images above were gathered from employees of the US Census Bureau and Secondary School students, consisting of approximately 250 individuals. There are 60,000 training images and 10,000 test images (sampled in a disjoint manner from the two sample groups). The images themselves are 28 × 28 unregistered grayscale images. Research has been performed on the dataset since the 1980s and has continued to the present day where it is often used in tutorials for neural net software frameworks.

This lecture will focus on this dataset as we investigate ways to train networks with large numbers of parameters, and how to adapt neural networks to structured input data such as images and in so doing dramatically reduce the number of parameters that need to be learned.
Training Neural Networks

As we progress beyond the simple datasets we have explored so far into more complicated domains, we encounter problems of scale. Take as an example the network below:

For networks of this size, the notation we used in the first lecture is a bit unwieldy. Since this is a feed-forward network, however, we can use a new notation that describes the math of the network using linear algebra. Given an input vector $\mathbf{x}$, we can succinctly write the output of this network as:

$$y = \mathbf{W}_2 S (\mathbf{W}_1 S (\mathbf{W}_0 x))$$

$$S(x) = \left\{ s_i \in \mathbb{R} | s_i = \frac{1}{1 + e^{-x_i}} \right\}$$

Similarly, we can compute all of the gradients for a layer at once:

$$\frac{\partial L}{\partial \mathbf{W}_1} = S_0(x) \left[ \frac{\partial S_1(x)}{\partial F_1} \odot \left( 1 - S_1(x) \right) \odot \frac{\partial \mathbf{W}_2^T}{\partial \mathbf{O}_2} \odot \varsigma \right] \odot \frac{\partial F_1}{\partial \mathbf{W}_1}$$

$$\varsigma = \left\{ \varsigma_i \in [0, 1] | \varsigma_i = P(i) - \delta_{di} \right\}$$
Problems of Scale

While this new notation is nice, it obscures the sheer scale of the inference happening. We can easily compute the number of parameters in this network:

$$\left| \{ w_{ijk} \in w | i = 0, j = 1 \} \right| = 784 \times 1000 = 784000$$
$$\left| \{ w_{ijk} \in w | i = 1, j = 2 \} \right| = 1000 \times 1000 = 1000000$$
$$\left| \{ w_{ijk} \in w | i = 2, j = 3 \} \right| = 1000 \times 10 = 10000$$
$$|w| = 1785000$$

This is quite a few more than the networks we originally explored. However, the problem of scale is not in just the number of trainable parameters in this network. We also have to contend with the 60,000 training images in the MNIST dataset. We can perform backpropagation for this entire dataset to compute the gradients of the log loss, but that would require computing $1785000 \times 60000 = 1.071 \times 10^{11}$ partial gradients in total at each update step. We need an alternative.
Stochastic Gradient Descent

As we recall from the previous lecture, our purpose in computing all of these gradients is in order to compute the derivative of the log loss with respect to the weights, where the log loss is computed as:

$$L(w, D) = -\frac{1}{D} \sum_{d} \sum_{i} \delta_{di} \log P(i|x_d, w)$$  \hspace{1cm} (1)

The important thing to notice in this equation is that the quantity we are interested in is actually the expected loss for an image. This means that we can estimate this value, which is the principle behind stochastic gradient descent. Instead of using the entire dataset every time, we can sample the dataset uniformly at random without replacement and use this smaller batch, or mini-batch to estimate the expected loss and compute our gradients. Each complete random walk through the training data is commonly called an epoch. What this means is that we can now compute the update for a particular weight as follows:

$$\nabla w_{ijk} = \eta \frac{\partial L(w, B)}{\partial w_{ijk}}$$  \hspace{1cm} (2)

where $\eta$ is the learning rate and $B \subset D$ is the mini-batch. This optimisation technique is called fixed rate stochastic gradient descent, but there are many variations upon this simple scheme that are often used.
Stochastic Gradient Descent, cont.

**Momentum** A very common means of augmenting stochastic gradient descent is by adding a momentum term, which alters the weight update in the following way:

$$w_{ijk}^{\tau+1} = w_{ijk}^\tau - \nabla w_{ijk}^\tau + \epsilon \nabla w_{ijk}^{\tau-1}$$  

where $\epsilon$ is called the *momentum* term. This has the effect of better convergence properties in “ravines” within the optimisation manifold, where it would otherwise oscillate.

**Stepped Learning Rate Updates** As the network converges to a local minimum, it is often necessary to make smaller adjustments in the weight values, which translates to smaller step sizes during weight updates. A simple way of doing this is by using a learning rate that changes over time in a stepped fashion:

$$\eta^\tau = \eta^0 \gamma^{\lfloor \tau/\sigma \rfloor}$$  

where the initial learning rate $\eta^0$ is multiplied by $\gamma$ every $\sigma$ steps.

**Inverse Learning Rate Updates** Another method is to continuously alter the learning rate in inverse proportion to the number of steps:

$$\eta^\tau = \frac{\eta^0}{(1 + \gamma \tau)^\rho}$$  

where $\gamma$ and $\rho$ control the speed of learning rate decay.
These variations on stochastic gradient descent all have the same problem: they set a global learning rate for all parameters. While we can alter the learning rate over time, the fundamental issue is that the individual parameters will need to change at different rates given changing local properties of the optimisation surface. A more recent approach called the Adam algorithm [1] overcomes this issue by storing parameter-specific moving averages of the first and second raw moments (\(m\) and \(v\) below). At each timestep, we update the weights in the following way:

\[
\begin{align*}
    g^\tau &= \frac{\partial L}{\partial w} \\
    m^\tau &= \beta_1 m^{\tau-1} + (1 - \beta_1) g^\tau \\
    v^\tau &= \beta_2 v^{\tau-1} + (1 - \beta_2) (g^\tau \odot g^\tau) \\
    \hat{m}^\tau &= \frac{m^\tau}{1 - (\beta_1)^\tau} \\
    \hat{v}^\tau &= \frac{v^\tau}{1 - (\beta_2)^\tau} \\
    w^\tau &= w^{\tau-1} - \alpha \frac{\hat{m}^\tau}{\sqrt{\hat{v}^\tau + \epsilon}}
\end{align*}
\]

The quantity \(\alpha \frac{\hat{m}^\tau}{\sqrt{\hat{v}^\tau + \epsilon}}\) has some useful properties. First, its upper bound is \(\alpha \frac{1 - \beta_1}{\sqrt{1 - \beta_2}}\), which removes the possibility of exploding gradients (a common issue with global learning rate methods). It also has the helpful property that as a parameter reaches an optimum the update value goes to zero. Some good values for the hyperparameters are: \(\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999\) and \(\epsilon = 10^{-8}\), which work for most problems.
Convolutional Neural Nets

Initial Results

Here is the network we looked at in the previous lecture:

![Network Diagram]

When we train it using stochastic gradient descent we get the following training curve:

![Training Curve]

where the network is presented with the training images and is then evaluated on an unseen test set. Here are some sample images and results:

<table>
<thead>
<tr>
<th>Image</th>
<th>Actual</th>
<th>Predicted</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Image]</td>
<td>![Actual]</td>
<td>![Predicted]</td>
</tr>
<tr>
<td>![Image]</td>
<td>![Actual]</td>
<td>![Predicted]</td>
</tr>
<tr>
<td>![Image]</td>
<td>![Actual]</td>
<td>![Predicted]</td>
</tr>
<tr>
<td>![Image]</td>
<td>![Actual]</td>
<td>![Predicted]</td>
</tr>
<tr>
<td>![Image]</td>
<td>![Actual]</td>
<td>![Predicted]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Image</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Predicted</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>
Demo

In this demo we will learn more about the simple network from the previous slide by examining its learned parameters and different feature responses when presented with sample data.

borfudin.github.io/anndemos/simple.html
Convolutional Neural Nets

Convolutional Layers

So far we have focused on artificial neural networks in which every node of one layer has an individual weighted connection to every node of the previous layer, but this is not necessary nor even optimal for some data. In the case of visual or aural signals, in which there is significant structure in the signal itself, we can use the same principles of convolution which have been covered in previous lectures to achieve the same (or better) accuracy with significantly fewer parameters.

We see above a very small image, two convolutional kernels, and their corresponding outputs. Each output node is computed as:

\[ o_{f,r,c} = w_{f00}i_{r-1,c-1} + w_{f01}i_{r-1,c} + w_{f02}i_{r-1,c+1} + w_{f10}i_{r,c-1} + w_{f11}i_{r,c} + w_{f12}i_{r,c+1} + w_{f20}i_{r+1,c-1} + w_{f21}i_{r+1,c} + w_{f22}i_{r+1,c+1} \]

With these layers the weights are being shared by multiple output nodes, and as such the total number of parameters is significantly less than in the fully-connected case. Being shared across multiple output nodes and being applied in a spatial manner can cause these weights to take on (in the first layer) the form of common low-level Gabor-like filters from computer vision which have been seen before.
The notation above is unwieldy, so ideally we would like to use the same linear algebra notation which we used for fully connected networks. It is possible, provided we first perform a simple transformation on $x$. We extract patches in row column order and embed them as columns in a matrix, as shown below:

$$x = \begin{pmatrix} a & b & c & d & e \\ f & g & h & i & j \\ k & l & m & n & o \\ p & q & r & s & t \\ u & v & w & x & y \end{pmatrix}$$

$$X = \begin{bmatrix} a & b & \ldots & g & \ldots & m \\ b & c & \ldots & h & \ldots & n \\ c & d & \ldots & i & \ldots & o \\ f & g & \ldots & l & \ldots & r \\ g & h & \ldots & m & \ldots & s \\ h & i & \ldots & n & \ldots & t \\ k & l & \ldots & q & \ldots & w \\ l & m & \ldots & r & \ldots & x \\ m & n & \ldots & s & \ldots & y \end{bmatrix}$$

We will denote this embedding operation as $X = \mathcal{E}(x)$. We can then store the weights in a weight matrix and use matrix multiplication to perform the convolution:

$$W_i = \begin{bmatrix} w_{000} & w_{001} & w_{002} & w_{010} & w_{011} & \ldots & w_{021} & w_{022} \\ w_{100} & w_{101} & w_{102} & w_{110} & w_{111} & \ldots & w_{121} & w_{122} \end{bmatrix}$$

$$C_i(x) = W_iX$$
Case Study: LeNet-5

We will look at a classic neural network named LeNet-5 after its inventor, Yann LeCun [2]. It has the following structure:

It contains four convolutional layers, two of which perform standard convolutions and two others which perform a parameterised sub-sampling. Finally, once the input image has been sufficiently processed it uses several fully connected layers to classify it. Below is the graph when trained using SGD for 20 epochs:

As can be seen, it matches the same accuracy as the fully connected network but with only 59,558 parameters as opposed to two million. Notice that the first layer features are not what one would expect:
Backpropagation

The weight sharing which takes place in convolutional layers makes the backpropagation process more complicated, but our new matrix notation will make things easier. First, however, we need to define another matrix manipulation:

\[
W_i = \begin{bmatrix}
w_{000} & w_{001} & w_{002} & w_{011} & \cdots & w_{021} & w_{022} \\
w_{100} & w_{101} & w_{102} & w_{111} & \cdots & w_{121} & w_{122} \\
w_{202} & w_{201} & w_{202} & w_{011} & \cdots & w_{001} & w_{000} \\
w_{212} & w_{211} & w_{212} & w_{111} & \cdots & w_{101} & w_{100}
\end{bmatrix}
\]

\[
W_i^R = \begin{bmatrix}
w_{022} & w_{021} & w_{020} & w_{012} & w_{011} & \cdots & w_{001} & w_{000} \\
w_{122} & w_{121} & w_{120} & w_{112} & w_{111} & \cdots & w_{101} & w_{100}
\end{bmatrix}
\]

By using the embedding function \( \mathcal{E} \) and the rotated matrix above we can compute all of the partial gradients for a layer in LeNet:

\[
\frac{\partial L}{\partial O_7} = \varsigma \quad \frac{\partial O_7}{\partial S_6} = W_7^T
\]

\[
\frac{\partial F_6}{\partial F_5} = W_6^T \quad \frac{\partial S_6}{\partial F_6} = S_6(x) \odot (1 - S_6(x))
\]

\[
\frac{\partial F_5}{\partial S_4} = W_5^T \quad \frac{\partial S_4}{\partial C_4} = S_4(x) \odot (1 - S_4(x))
\]

\[
\frac{\partial C_4}{\partial C_3} = W_4^R \quad \frac{\partial C_3}{\partial W_3} = S_2(x)
\]

\[
\frac{\partial L}{\partial W_3} = \frac{64 \times 100}{16 \times 64} \mathcal{E}^{-1} \begin{bmatrix}
\frac{\partial S_4}{\partial C_4} & \frac{\partial F_3}{\partial S_4} & \frac{\partial F_6}{\partial S_6} & \frac{\partial O_7}{\partial O_7} & \frac{\partial L}{\partial L}
\end{bmatrix} \begin{bmatrix}
\mathcal{E}(S_2(x))^T
\end{bmatrix} \begin{bmatrix}
\frac{100 \times 150}{1176 \times 1}
\end{bmatrix}
\]
Demo

In this demo we will learn more about the LeNet-5 network by examining its learned parameters and different feature responses when presented with sample data.

borfudin.github.io/anndemos/lenet_mnist.html
Vanishing Gradients

We started with a single perceptron, and then connected them in layers to create a multi-layer perceptron. We have now added a new kind of layer and created a network with seven layers and trained it via stochastic gradient descent. In doing so we have now reached the state of the art in the 1990s, when research into neural nets entered into a second doldrums due to hardware limitations and the problem of the vanishing gradient. Here are the two most common activation functions in early ANNs:

\[
\text{tanh} \ x = \frac{e^x - e^{-x}}{e^x + e^{-x}}
\]

\[
\frac{d}{dx} \text{tanh} \ x = 1 - \text{tanh}^2 \ x
\]

\[
\text{s}(x) = \frac{1}{1 + e^{-x}}
\]

\[
\frac{d}{dx} \text{s}(x) = \text{s}(x) (1 - \text{s}(x))
\]

For both, the value of the function and thus the derivative is between 0 and 1. Since all the non-linearities in a network are multiplied together during backpropagation, the gradient becomes smaller the farther backward it is propagated. For deep networks, it becomes so small that it causes convergence to slow down or stop entirely.

The next lecture will explore how key innovations around non-linearities and neural network structure combined with advances in computer hardware to bring about the phenomenon of deep learning, which has revolutionised modern machine learning and computer vision.
References
