3b: Image Processing

Weight Initialization

The aim of weight initialization is to choose the size of the initial weights in each layer of a deep neural network in such a way that the gradients will remain in a healthy range and not vanish or explode.

Consider a neural network with D layers and let the activations at each layer (i) be $x^{(i)} = \{x_k^{(i)}\}_{1\le k\le n_{(i)}}$ where $x = x^{(1)}$ is the input and $z = x^{(D+1)}$ is the output. If $w_{jk}^{(i)}$ is the weight connecting node k in layer (i) to node j in layer ($i+1$), then

$$x_j^{(i+1)} = g\big(\sum_{k=1}^{n_{(i)}} w_{jk}^{(i)} x_k^{(i)}\big)$$

where $g()$ is the transfer function. If we assume that the activations (and the weights) are statistically independent and identically distributed (i.i.d.) we can estimate the mean and variance of $x^{(i+1)}$ based on those of $x_k^{(i)}$ and $w_{jk}^{(i)}$. The bias weights are initialised to 0, and we also assume (for now) that the activation function is $\tanh()$ which is radially symmetric about the origin.

Recall that the **mean** and **variance** of a set of n samples $x_1, \ldots, x_n$ are given by

$$\begin{aligned}\mathrm{Mean}[x] &= \frac{1}{n}\sum_{k=1}^{n} x_k \\ \mathrm{Var}[x] &= \frac{1}{n}\sum_{k=1}^{n}(x_k - \mathrm{Mean}[x])^2 = \left(\frac{1}{n}\sum_{k=1}^{n} x_k^2\right) - \mathrm{Mean}[x]^2\end{aligned}$$

If $s = \sum_{k=1}^{n} x_k$ where x_k are i.i.d. then $\mathrm{Mean}[s] = n\,\mathrm{Mean}[x]$ and $\mathrm{Var}[s] = n\,\mathrm{Var}[x]$.

More generally, if $y = \sum_{k=1}^{n} w_k x_k$ where w_k, x_k are i.i.d. with $\mathrm{Mean}[w] = \mathrm{Mean}[x] = 0$ then

$$\mathrm{Var}[y] = n\,\mathrm{Var}[w]\,\mathrm{Var}[x]$$

Statistics Example: Coin Tossing

Suppose we toss a coin once, and count the number of heads. The mean and variance of this value are

$$\mu = \frac{1}{2}(0+1) = 0.5$$
$$\sigma^2 = \frac{1}{2}((0-0.5)^2 + (1-0.5)^2)) = 0.25$$

If we toss the coin 100 times, the mean and variance will be $\mu = 100 \times 0.5 = 50$ and $\sigma^2 = 100 \times 0.25 = 25$. If we toss it 10,000 times, they will be $\mu = 10,000 \times 0.5 = 5000$ and $\sigma^2 = 10,000 \times 0.25 = 2500$.

Note that instead of the variance we often think in terms of the **standard deviation**, which in this case would be $\sigma = 0.5, 5$ and 50, respectively. This means that the number of heads as a **fraction** of the total number of coin tosses will get steadily closer to 0.5 as the number of tosses increases.

Weight Initialisation

Returning to our neural network example, we have

$$x_j^{(i+1)} = g\left(\sum_{k=1}^{n_{(i)}} w_{jk}^{(i)} x_k^{(i)}\right)$$
$$\mathrm{Var}\left[\sum_{k=1}^{n_{(i)}} w_{jk}^{(i)} x_k^{(i)}\right] = n_{(i)}\,\mathrm{Var}\left[w^{(i)}\right]\mathrm{Var}\left[x^{(i)}\right]$$
$$\text{So}\quad \mathrm{Var}\left[x^{(i+1)}\right] \simeq G_0 n_{(i)}\,\mathrm{Var}\left[w^{(i)}\right]\mathrm{Var}\left[x^{(i)}\right]$$

where G_0 is a constant whose value is estimated to take account of the transfer function. By multiplying across all D layers from input $x = x^{(1)}$ to output $z = x^{(D+1)}$, we get

$$\mathrm{Var}[z] \simeq \left(\prod_{i=1}^{D} G_0 n_{(i)}\,\mathrm{Var}\left[w^{(i)}\right]\right)\mathrm{Var}[x]$$

When we apply gradient descent through backpropagation, the differentials will follow a similar pattern:

$$\mathrm{Var}\left[\frac{\partial}{\partial x}\right] \simeq \left(\prod_{i=1}^{D} G_1 n_{(i+1)}\,\mathrm{Var}\left[w^{(i)}\right]\right)\mathrm{Var}\left[\frac{\partial}{\partial z}\right]$$

Where G_1 is a constant whose value is estimated to take account of the derivative of the transfer function. If some layers are not fully connected, we can replace $n_{(i)}$ in the above equations with the average number of incoming connections to each node at layer (i+1), and replace $n_{(i+1)}$ with the average number of outgoing connections from each node at layer (i).

In order to have healthy forward and backward propagation, we need to choose the initial weights

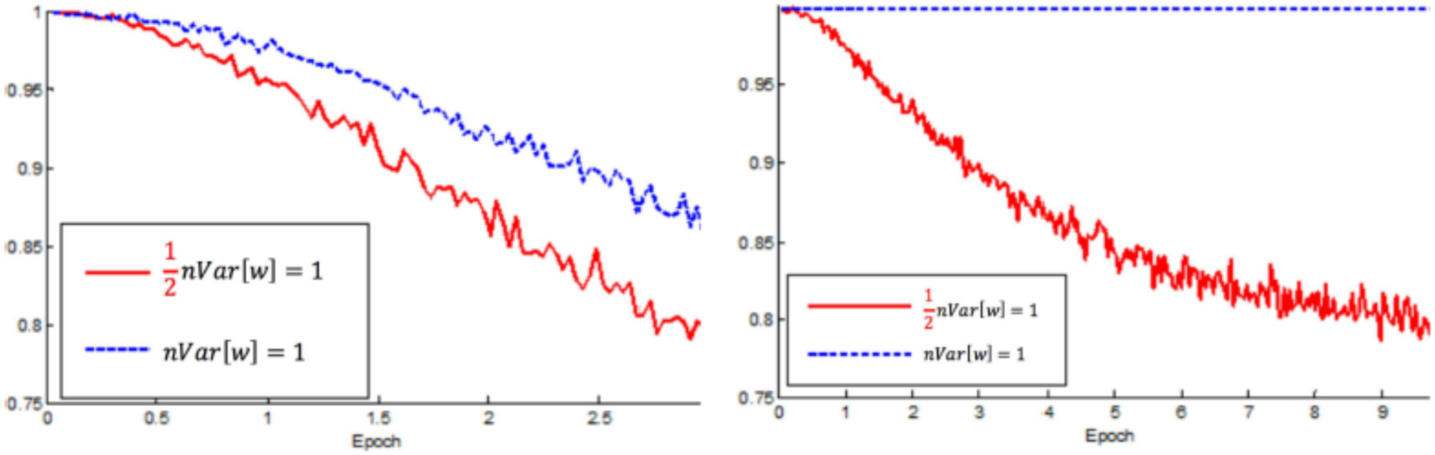
$\{w_{jk}^{(i)}\}$ in each layer (i) such that all terms in the product are approximately equal to 1. Any deviation from this could cause the differentials to either decay or explode exponentially. If the transfer function is $\tanh()$, this is normally achieved using **xavier** initialisation, where the weights are chosen from a uniform distribution bounded between these values:

$$\pm\frac{\sqrt{6}}{\sqrt{n_{(i)}+n_{(i+1)}}}$$

For Rectified Linear Units (ReLU) the above analysis is essentially still valid, with $G_0 = G_1 = \frac{1}{2}$, although we need to be mindful of the fact that the mean of the activations $x^{(i)}$ is not zero. In this case we normally use **kaiming** initialisation, where the weights are chosen from a Gaussian distribution with mean 0 and standard deviation

$$\frac{\sqrt{2}}{\sqrt{n_{(i)}}}$$

These figures from (He et al., 2015) illustrate the benefit of kaiming initialisation, on the ImageNet classification task.



22-layer ReLU network (left), $\mathrm{Var}[w] = \frac{2}{n}$ converges faster than $\mathrm{Var}[w] = \frac{1}{n}$
30-layer ReLU network (right) $\mathrm{Var}[w] = \frac{2}{n}$ is successful while $\mathrm{Var}[w] = \frac{1}{n}$ fails to learn at all.

Optional video

References

He, K., Zhang, X., Ren, S., & Sun, J., 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, *Proceedings of the IEEE International Conference on Computer Vision* (pp. 1026-1034).

Batch Normalization

Batch Normalisation (Ioffe & Szegedy, 2015) serves a similar purpose to weight Initialisation, but is applied throughout the training process rather than just at the beginning.

The activations $x_k^{(i)}$ of node k in layer (i) can be normalised relative to their mean and variance across a mini-batch of training items:

$$\hat{x}_k^{(i)} = \frac{x_k^{(i)} - \text{Mean}\big[x_k^{(i)}\big]}{\sqrt{\text{Var}\big[x_k^{(i)}\big]}}$$

These activations can then be shifted and re-scaled to have mean $\beta_k^{(i)}$ and standard deviation $\gamma_k^{(i)}$

$$y_k^{(i)} = \beta_k^{(i)} + \gamma_k^{(i)} \hat{x}_k^{(i)}$$

Inspired by Weight Initialisation, we might at first think that $\beta_k^{(i)}$ and $\gamma_k^{(i)}$ should be fixed in advance (for example, if $\beta_k^{(i)} = 0$ and $\gamma_k^{(i)} = 1$, then $y_k^{(i)}$ would be equal to $\hat{x}_k^{(i)}$). However, it turns out that better results can be obtained if $\beta_k^{(i)}$ and $\gamma_k^{(i)}$ are treated as additional parameters for each node, which can be trained by backpropagation along with the other parameters (weights) in the network. In this way, the network retains the same flexibility it would have had without Batch Normalization, but the dynamics of he backpropagation are changed in a beneficial way.

After training is complete, $\text{Mean}\big[x_k^{(i)}\big]$ and $\text{Var}\big[x_k^{(i)}\big]$ can either be pre-computed on the entire training set, or updated using running averages.

Optional video

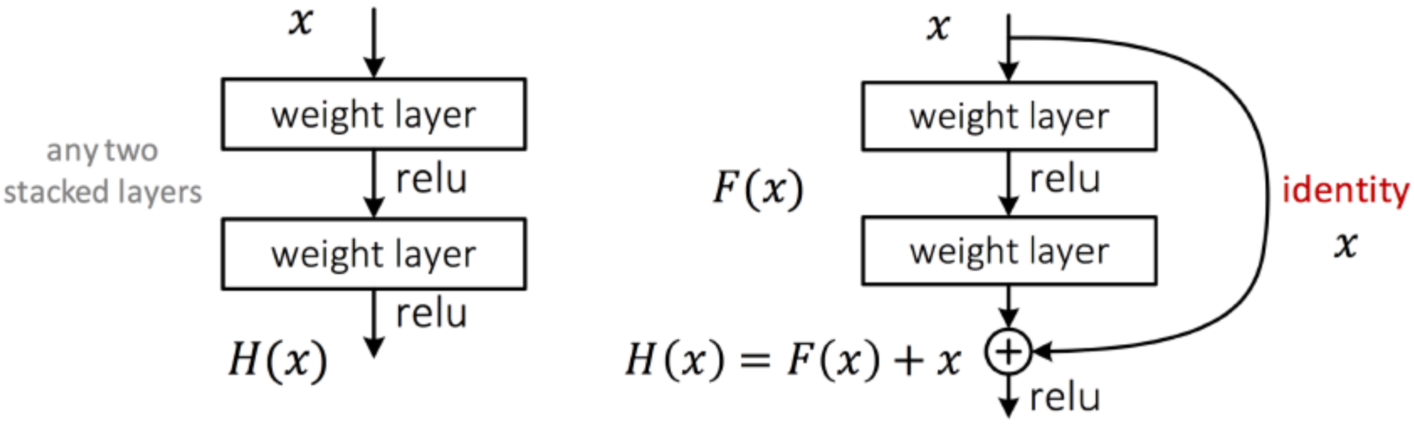
References

Ioffe, S., & Szegedy, C., 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning* (pp. 448-456).

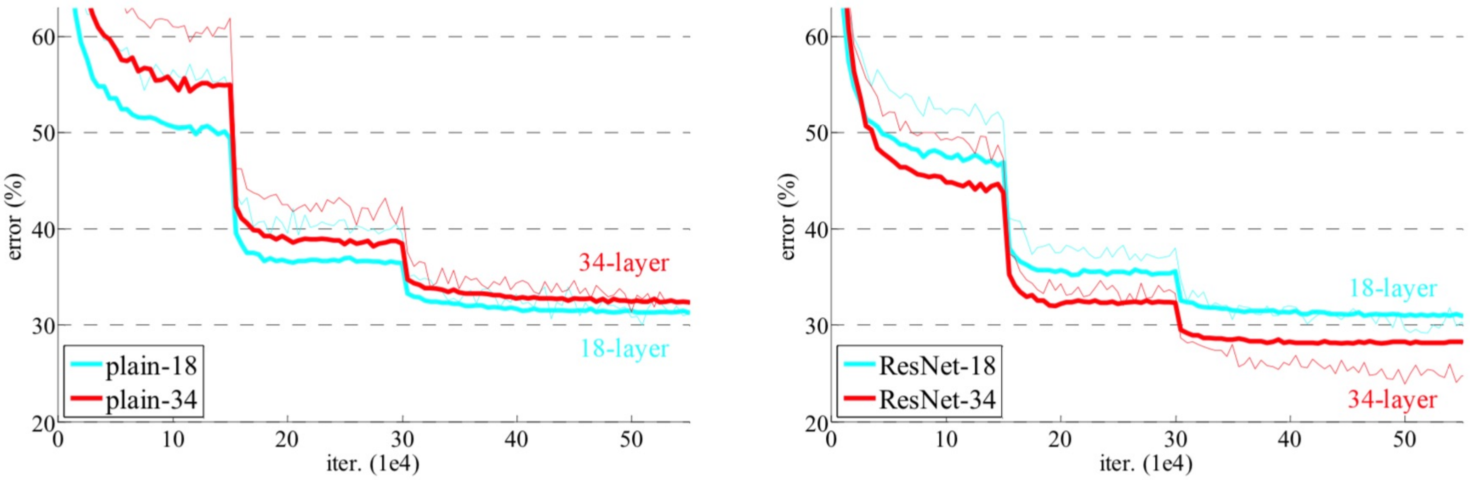
ResNets and DenseNets

Residual Networks

Very deep networks (greater than 30 layers) can be trained successfully by introducing **skip connections** to form a residual network. The idea is to take any two consecutive stacked layers in a deep network and add a "skip" connection which bypasses these layers and is added to their output.



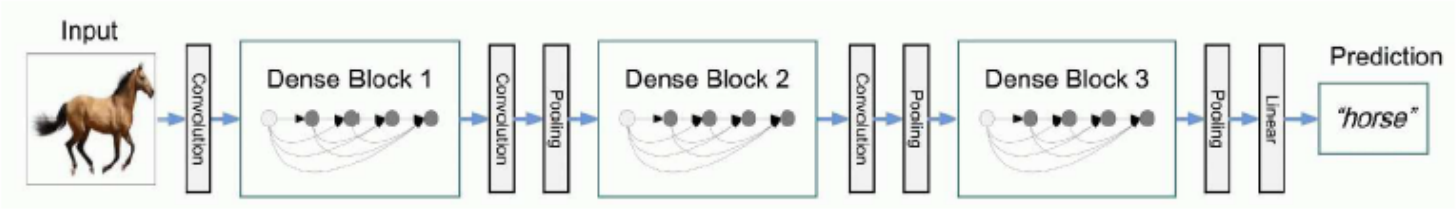
In this way, the preceding layers attempt to do the "whole" job, making x as close as possible to the target output of the entire network. $F(x)$ is a residual component which corrects the errors from previous layers, or provides additional details which the previous layers were not powerful enough to compute.



These graphs from (He, 2016) demonstrate the effectiveness of residual networks. When the skip connections are absent (left) the test error for a 34-layer network is higher than for an 18-layer network. When the skip connections are included (right) both the training error (thin) and test error (thick) are lower for the 34-layer network than for the 18-layer network.

In order to train a network with more than 100 layers, the transfer function (ReLU) needs to be applied before adding the residual instead of afterwards. This is called an **identity skip connection**.

Dense Networks



Good results on ImageNet have also been achieved using networks with densely connected blocks. Within each block, every layer is connected by shortcut connections to all the preceding layers (Huang, 2017).

Optional video

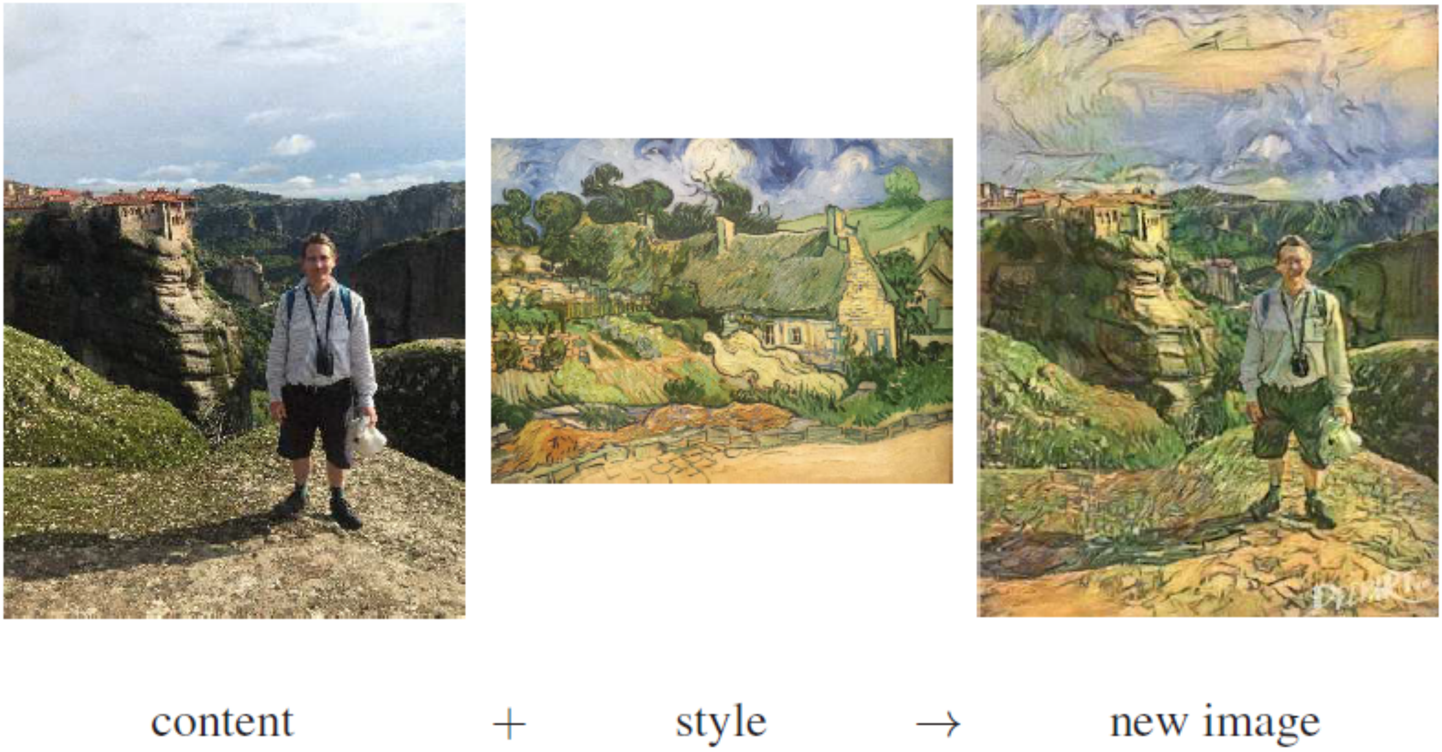
References

He, K., Zhang, X., Ren, S., & Sun, J., 2016. Deep Residual Learning for Image Recognition, In *Proceedings of the IEEE Conference on computer Vision and Pattern Recognition* (pp770-778).

Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K.Q., 2017. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 4700-4708).

Neural Style Transfer

Convolutional Neural Networks can be used for many tasks other than object classification. One such task is Neural Style Transfer (Gatys, 2016), which aims to combine the content of one image with the style of another, as shown here.



The process relies on a fixed CNN such as VGG-19 which has been pre-trained on ImageNet. If F^l_{ik} denotes the activation of the i^{th} convolutional filter at spatial location k in layer l, then it is natural to minimise the L$_2$ distance between $F^l_{ik}(x)$ and $F^l_{ik}(x_c)$, where x_c is the content image and x is the synthetic image being generated. Moreover, classical work on texture synthesis would suggest that the visual "style" of an image is somehow captured in the Gram matrices

$$G^l_{ij} = \sum_k F^l_{ik} F^l_{jk}$$

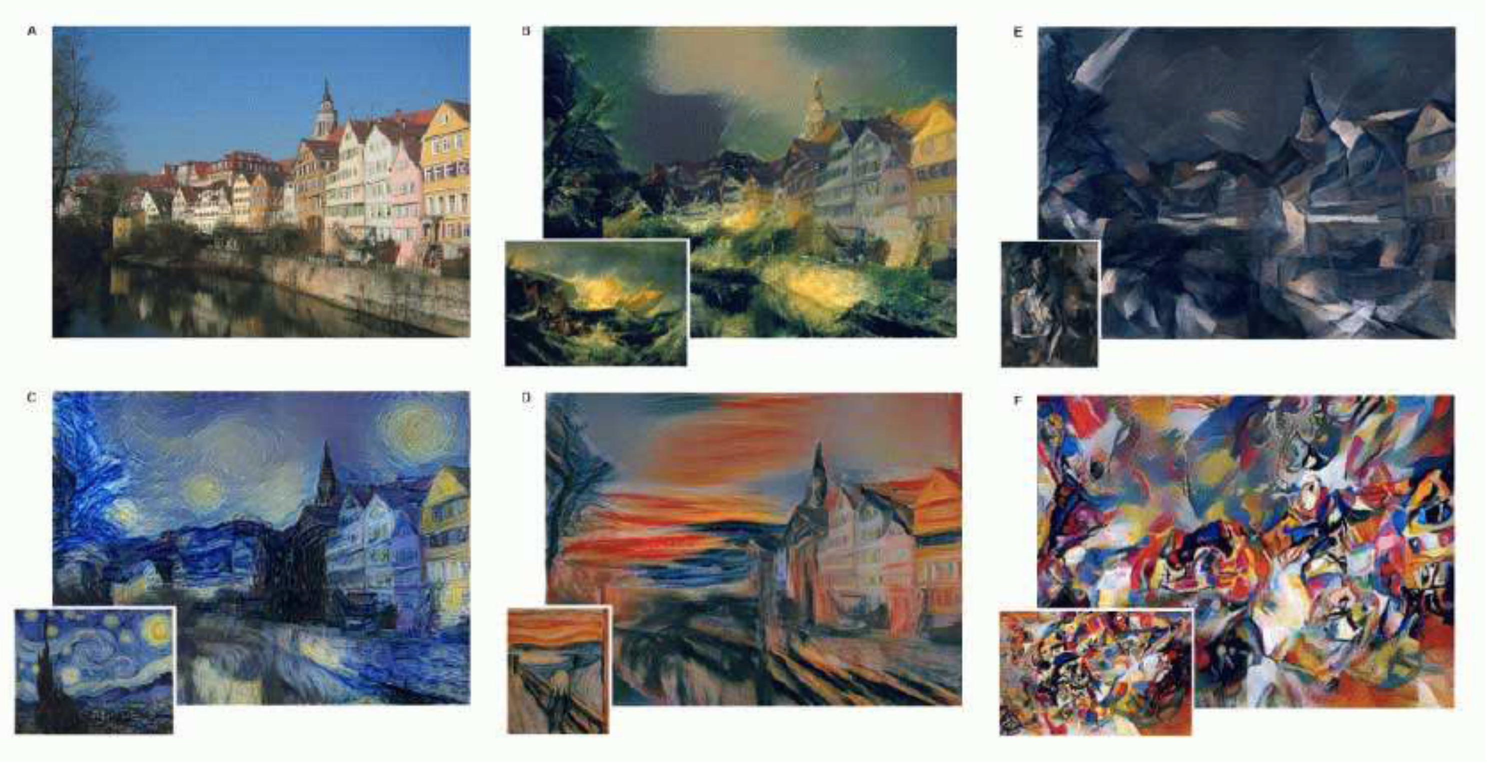
Neural Style Transfer therefore aims to minimise this loss function:

$$\begin{aligned} E_{\text{total}} &= E_{\text{content}} + E_{\text{style}} \\ &= \frac{\alpha}{2} \sum_{i,k} ||F^l_{ik}(x) - F^l_{ik}(x_c)||^2 + \frac{\beta}{4} \sum_{l=0}^{L} \frac{w_l}{N_l^2 M_l^2} \sum_{i,j} \left(G^l_{ij} - A^l_{ij}\right)^2 \end{aligned}$$

where x is the generated image, x_c is the content image, F^l_{ik} is the ith filter at position k in layer l, N_l, M_l are the number of filters and size (area) of the hidden layer, w_l is a weighting factor for layer l

, and G^l_{ij}, A^l_{ij} are the Gram matrices for the style image and the generated image.

Note that in this case, gradient descent is applied not to the weights of the network (which remain fixed) but rather to the R,G,B values of the pixels in the image itself. This figure from Gatys (2016) shows a single content image in combination with five different style images.



Optional video

References

Gatys, L.A., Ecker, A.S., & Bethge, M. 2016. Image style transfer using convolutional neural networks, In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 2414-2423).

Revision 5: Image Processing

This is a revision quiz to test your understanding of the material from Week 3 on image processing.

You must attempt to answer each question yourself, before looking at the sample answer.

Question 1 *Submitted Sep 17th 2022 at 3:15:15 pm*

Explain the problem of vanishing and exploding gradients, and how Weight Initialization can help to prevent it.

Solution

The differentials in a deep neural network tend to grow according to this equation

$$\mathrm{Var}\left[\frac{\partial}{\partial x}\right] \simeq \left(\prod_{i=1}^{D} G_1 n_{(i+1)} \mathrm{Var}\left[w^{(i)}\right]\right) \mathrm{Var}\left[\frac{\partial}{\partial x}\right]$$

where $w^{(i)}$ are the weights at layer i, $n_{(i+1)}$ is the number of weights fanning out from each node in layer i, and G_1 estimates the average value of the derivative of the transfer function. If the weights are initialized so that the factor in parentheses corresponding to each layer is approximately 1, then the differentials will remain in a healthy range. Otherwise, they may either grow or vanish exponentially.

Question 2 *Submitted Sep 17th 2022 at 3:14:02 pm*

Describe the Batch Normalization algorithm.

Solution

The mean and variance of the activations $x_k^{(i)}$ at layer i over a batch of training items are estimated or pre-computed, and *normalized* activations are calculated for each node

$$\hat{x}_k^{(i)} = \frac{\hat{x}_k^{(i)} - \mathrm{Mean}\left[x_k^{(i)}\right]}{\sqrt{\mathrm{Var}\left[x_k^{(i)}\right]}}$$

These activations are then shifted and rescaled by

$$y_k^{(i)} = \beta_k^{(i)} + \gamma_k^{(i)} \hat{x}_k^{(i)}$$

where $\beta_k^{(i)}, \gamma_k^{(i)}$ are additional parameters to be learned by backpropagation.

Question 3 *Submitted Sep 17th 2022 at 3:13:51 pm*

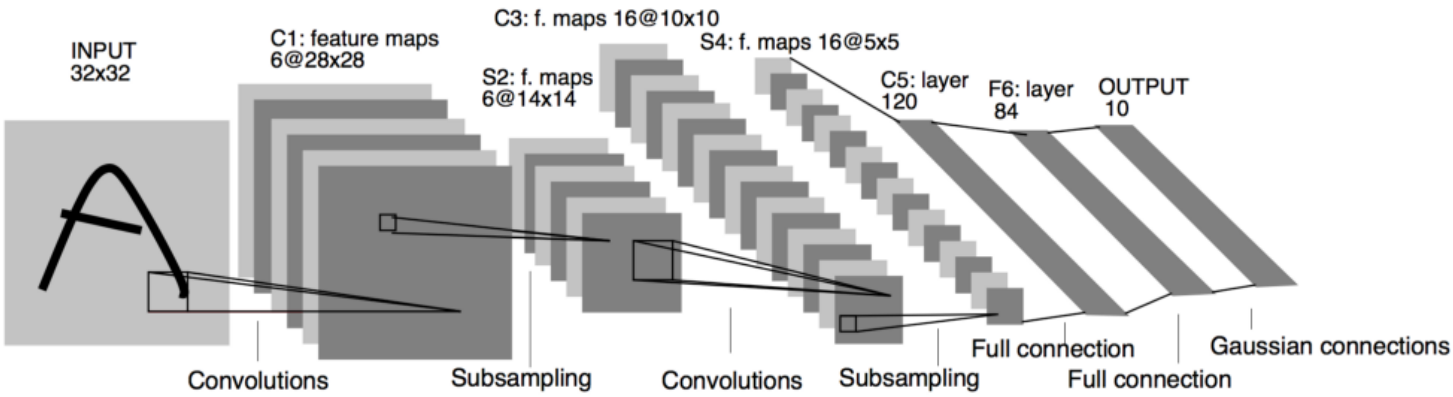
Explain the difference between a Residual Network and a Dense Network.

A Residual Network includes "skip" connections which bypass each pair of consecutive layers. These intermediate layers therefore compute a residual component, which is added to the output from previous layers and corrects their errors, or provides additional details which they were not powerful enough to compute.

A Dense Network is built from densely connected blocks, separated by convolution and pooling layers. Within a dense block, each layer is connected by shortcut connections to all preceding layers.

Coding Exercise: Image Classification

In this exercise, you will practice how to use PyTorch to create a neural network model named LeNet-5, one of the most famous convolutional neural network models proposed by Yann LeCun et al. in 1989. Please implement LeNet-5 according to the following description:



LeNet-5 consists of seven layers:

layer 1: Convolution, input channel = 1, output channel = 6, kernel size = 5, activation = ReLu.

layer 2: Max Pooling, kernel size = 2.

layer 3: Convolution, input channel = 6, output channel = 16, kernel size = 5, activation = ReLu.

layer 4: Max Pooling, kernel size = 2.

layer 5: Linear, output size 120, activation = ReLu. (Calculate the input size by yourself).

layer 6: Linear, input size 120, output size 84, activation = ReLu. (Calculate the input size by yourself).

layer 7: Linear, input size 84, output size 10, no activation.

Please **download** the notebook and implement/run it locally.