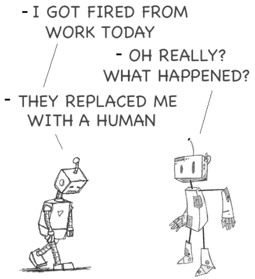
Deep Learning

Krystian Mikolajczyk & Carlo Ciliberto

Department of Electrical and Electronic Engineering

Imperial College London

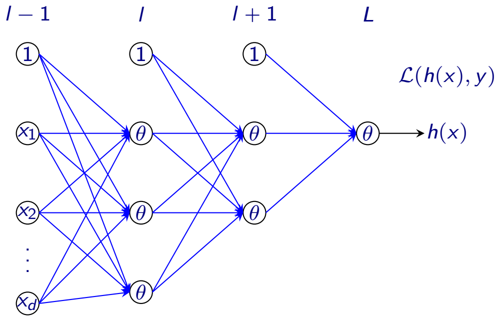

- I GOT FIRED FROM WORK TODAY
- OH REALLY? WHAT HAPPENED?
- THEY REPLACED ME WITH A HUMAN

Overview

- Backpropagation & problems
- Optimization
- Regularisation

Backpropagation

- Weights $W^{(l)}$, $w_{ij}^{(l)}$ $\begin{cases} 1 \leqslant l \leqslant L \text{ layers;} \\ 0 \leqslant i \leqslant d^{(l-1)} \text{ inputs;} \\ 1 \leqslant j \leqslant d^{(l)} \text{ outputs} \end{cases}$
- Input $\mathbf{x}$ is applied to the input layer
  $x_1^{(0)}, \dots, x_{d^{(0)}}^{(0)} \quad \Rightarrow \quad x_1^{(L)} = h(\mathbf{x}) \in \mathbb{R}$
- Activations (outputs) $x_j^{(l)} = \theta(s_j^{(l)}) = \theta\left(\sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)}\right)$
- Diagonal matrix of activations at layer l
  $\Theta(s^{(l)}) = \Theta(W^{(l)}\mathbf{x}^{(l-1)}) \in \mathbb{R}^{d^{(l)} \times d^{(l)}}$
- diagonal matrix of activation derivatives evaluated at $s^{(l)}$
  $\Theta'(s^{(l)}) \in \mathbb{R}^{d^{(l)} \times d^{(l)}}$
- Gradient w.r.t. $s^{(l)} \Rightarrow \delta^{(l)} = \Theta'(s^{(l)})(W^{(l)})^T \delta^{(l+1)} \in \mathbb{R}^{(l)}$
  $\delta^{(L)} = \Theta'(s^{(L)}) \nabla_{x^{(L)}} \mathcal{L} \in \mathbb{R}$



Gradient

$\delta^{(l)} = \left(\prod_{k=l}^{L-1} \Theta'(s^{(k)})(W^{(k)})^T\right) \Theta'(s^{(L)}) \nabla_{x^{(L)}} \mathcal{L}$

weights update $\Delta W^{(l)} = \nabla_W \mathcal{L} = -\eta x^{(l-1)} \delta^{(l)}$

- Requires forward propagation for error $\mathcal{L}$

Vanishing and exploding gradients

The issue of vanishing and exploding gradients stem from gradient backpropagation formula

$$\delta^{(l)} = \left(\prod_{k=l}^{L-1} \Theta'(s^{(k)})(W^{(k)})^T\right) \Theta'(s^{(L)}) \nabla_{x^{(L)}} \mathcal{L}$$

- Some activation functions (sigmoid, tanh) can saturate
- Small activations lead to slow-learning weights
- Deep networks can lead to vanishing or exploding gradients if weight matrices have small or large eigenvalues

Vanishing and exploding gradients

These problems are tackled through network design, initialisation and regularisation strategies

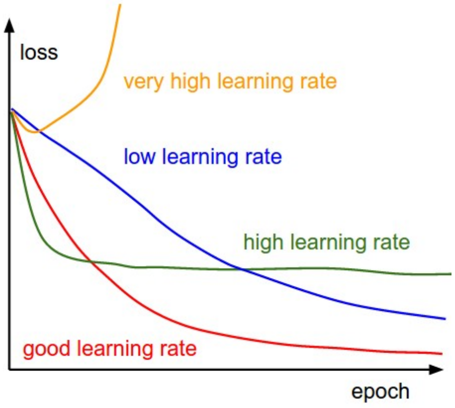
Learning rate

SGD (mini batch)

- Loss $\mathcal{L} = \frac{1}{n}\sum_i^n \ell(h(x_i), y_i)$
- Loss derivative $\nabla_W \mathcal{L} = \frac{1}{n}\sum_i^n \nabla_W \ell(h(x_i), y_i)$
- Weight update $W_{t+1} = W_t - \eta \nabla_W \mathcal{L}_t$

Learning rate η

- Reduces redundancy in gradient computation
- Faster (and usually better) convergence
- Can be changed online
- Another hyperparameter to tune



- Convergence with SGD can be very slow
- Setting the learning rate can be difficult, and often involves trial and error
- Learning rate schedules can be used to reduce the learning rate over the course of training different rates of learning
- Various optimisation algorithms have been proposed to address these problems

Epoch, Batch, Iterations

Epoch

- One Epoch is when an ENTIRE dataset is passed forward and backward through the neural network only ONCE
- Updating the weights with single pass through entire dataset or one epoch is not enough (leads to underfitting)
- As the number of epochs increases, the weight are changed in the neural network and the curve goes from underfitting to optimal to overfitting curve
- The right numbers of epochs is different for different datasets

Batch

- Entire dataset is divided into a number of batches or sets or parts, as entire dataset cannot be passed into the network at once
- Batch size is the total number of training examples present in a single batch
- Batch size and number of batches are two different things

Iteration

- Iterations is the number of batches needed to complete one epoch
- The number of batches is equal to number of iterations for one epoch

Overfitting

Typical reasons

- Network is too big
- Training for too long
- Not enough data

Remedy

- Reduce the network complexity (layers)
- Regularisation
    - Momentum and weight decay
    - Dropout
    - Weight initialisation
    - Batch Normalisation
- Data Augmentation
- Patience/early stopping
- Weight sharing
- Ensemble predictions (Pattern Recognition)
- Multitask learning
- Adversarial training (hard negatives)

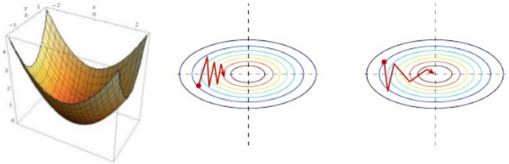
Optimizers

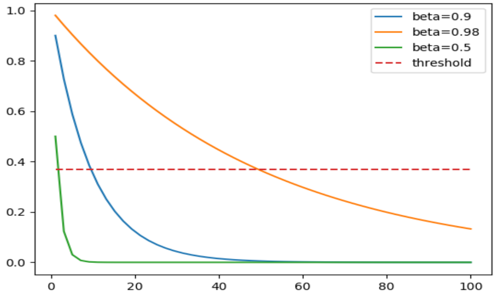
SGD with momentum (mini batch)

- Loss $\mathcal{L} = \frac{1}{n}\sum_i^n \ell(h(x_i), y_i)$
- Loss derivative $\nabla_W \mathcal{L} = \frac{1}{n}\sum_i^n \nabla_W \ell(h(x_i), y_i)$
- Momentum based update $Z_{t+1} = \beta Z_t + \nabla_W \mathcal{L}(W_t)$
- Weight update $W_{t+1} = W_t - \eta Z_{t+1}$

Momentum β

- Momentum accumulates (smooths) with past updates
- Accelerates convergence of SGD
- Typical β values is around 0.9
- Setting $\beta = 0$ reduces to standard SGD
- Another hyperparameter to tune





Optimizers

SGD with Nesterov momentum

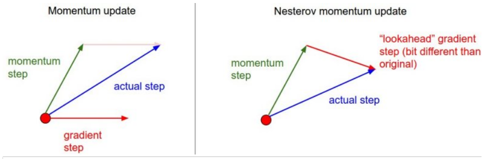
- Loss $\mathcal{L} = \frac{1}{n}\sum_i^n \ell(h(x_i), y_i)$
- Loss derivative $\nabla_W \mathcal{L}(W) = \frac{1}{n}\sum_i^n \nabla_W \ell(h(x_i), y_i)$
- Nesterov momentum update

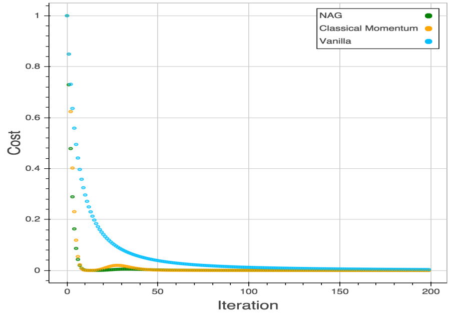
  $Z_{t+1} = \beta Z_t + \nabla_W \mathcal{L}(W_t - \eta\beta Z_t)$

  Basic momentum $Z_{t+1} = \beta Z_t + \nabla_W \mathcal{L}(W_t)$
- Weight update $W_{t+1} = W_t - \eta Z_{t+1}$

Nesterov momentum β

- 1. Make a step in the direction of the accumulated gradient
- 2. Measure the gradient in this new point and correct
- Nesterov Accelerates Gradient





Optimizers

Adagrad

- The update rule is (division and square root performed element-wise) $W_{t+1} = W_t - \frac{\eta}{\sqrt{G_t + \varepsilon}} \nabla_W \mathcal{L}(W_t)$ $G_t \in \mathbb{R}^{p \times p}$ - is a diagonal matrix where the diagonal elements are the sum of squares of gradients with respect to w_i up to iteration t

- Adapts the learning rate for each parameter
- Less frequent (active) parameters receive larger updates
- Well suited to sparse data
- Used to train GloVe word embeddings (text representation)
- The resulting learning rates per parameter are monotonically decreasing, and eventually the algorithm effectively stops learning

Optimizers

RMSProp

- The update rule is (division and square root performed element-wise)

$$W_{t+1} = W_t - \frac{\eta}{\sqrt{\mathbb{E}\left[(\nabla\mathcal{L})^2\right]_t + \varepsilon}} \odot \nabla\mathcal{L}_t$$

$\odot$ - is Hadamard product (element-wise)

$$\mathbb{E}\left[(\nabla\mathcal{L})^2\right]_t = \gamma\mathbb{E}\left[(\nabla\mathcal{L})^2\right]_{t-1} + (1-\gamma)(\nabla\mathcal{L}_t)^2$$

Adadelta

- The update rule is (division and square root performed element-wise)

$$W_{t+1} = W_t - \frac{\sqrt{\mathbb{E}\left[(\Delta W)^2\right]_t + \varepsilon}}{\sqrt{\mathbb{E}\left[(\nabla\mathcal{L})^2\right]_t + \varepsilon}} \odot \nabla\mathcal{L}_t$$

$$\mathbb{E}\left[(\Delta W)^2\right]_t = \gamma\mathbb{E}\left[(\Delta W)^2\right]_{t-1} + (1-\gamma)(W_t - W_{t-1})^2$$

- Aim to resolve the vanishing learning rates of Adagrad
- Uses a decaying average of past squared gradients
- γ is typically set similar to momentum (i.e. 0.9)
- Adadelta: Removes the need to set a default learning rate η

Optimizers

Adam (Adaptive moment estimation, RMSprop+momentum)

- The update rule is (division and square root performed element-wise)

$$W_{t+1} = W_t - \frac{\eta}{\sqrt{\mathbf{v}_t + \varepsilon}} \odot \mathbf{m}_t$$

$$\mathbf{m}_t = \mathbb{E}\left[(\nabla\mathcal{L})\right]_t / (1-\beta_1) \qquad \mathbb{E}\left[(\nabla\mathcal{L})\right]_t = \beta_1 \mathbb{E}\left[(\nabla\mathcal{L})\right]_{t-1} + (1-\beta_1)(\nabla\mathcal{L}_t)$$

$$\mathbf{v}_t = \mathbb{E}\left[(\nabla\mathcal{L})^2\right]_t / (1-\beta_2) \qquad \mathbb{E}\left[(\nabla\mathcal{L})^2\right]_t = \beta_2 \mathbb{E}\left[(\nabla\mathcal{L})^2\right]_{t-1} + (1-\beta_2)(\nabla\mathcal{L}_t)^2$$

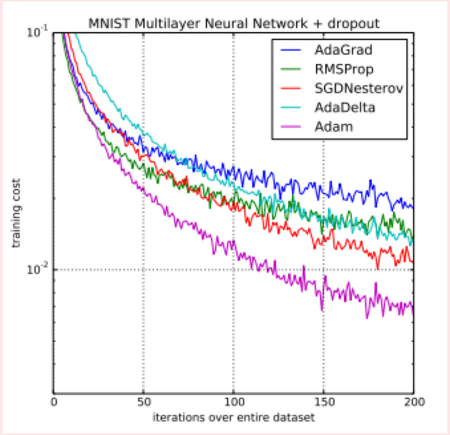
- Estimates first and second moments of the gradients
- Adapts the learning rate for each parameter
- m_t and v_t correct for an initial bias towards zero
- Typical decay parameters are $\beta_1 \approx 0.9$ and $\beta_2 \approx 0.999$ and $\varepsilon \approx 10^{-8}$

Optimizers

Summary

From top to bottom optimizers: Adam, RMSProp, Nestrov momentum, Adadelta, Adagrad

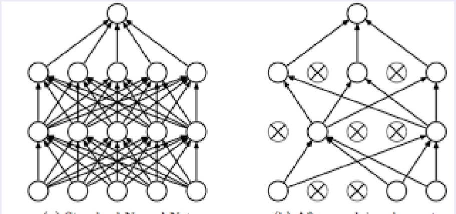
- Adagrad introduces adaptive learning rate; best suited for sparse data
- RMSProp resolves the vanishing learning rates by using decaying averages
- Adadelta is similar to RMSProp but does not require a learning rate
- Adam adds bias-correction and momentum
- SGD is still often used and tends to find a good minimiser and generalise well
- AdaMax (generalisation to L_p norm), Nadam (Adam+NAG), AMSGrad (max instead of exponential average)



Regularisation

Dropout

- Training: ignore (zero out) a random fraction p, of nodes (and corresponding activations) for each hidden layer, for each training sample, for each iteration
- Testing: use all activations, but reduce them by a factor p (to account for the missing activations during training)

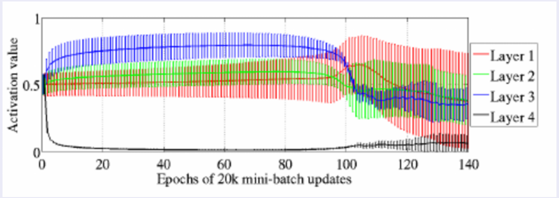


- Forces a neural network to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.
  - Doubles the number of iterations required to converge but training time for each epoch is shorter.
- With $|W|$ hidden units (each can be dropped), there are $2^{|W|}$ possible training models but only one test model
- FC layers only - generally less effective at regularizing convolutional layers.
  - CNN layers have few parameters, hence need less regularisation.
  - because of the spatial relationships encoded in feature maps, activations can become highly correlated.
- Non FC layer can use batch normalisation

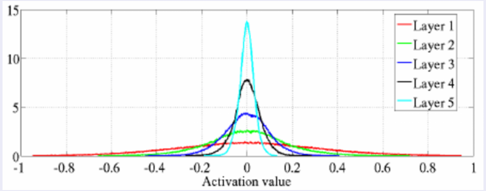
Regularisation

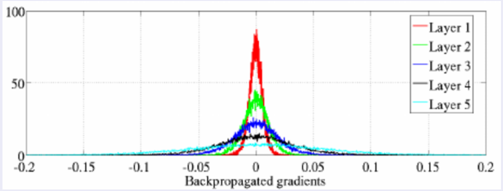
Varied neuron input/output distribution problem

- Mean and standard deviation of activation values.
  - Note the quick saturation of the top layer.



- Activation & gradient histograms for unnormalised distributions.
  - Note changing values with layers

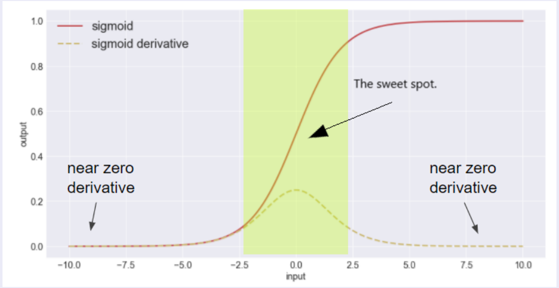




Regularisation

Varied neuron input/output distribution problem

- During training weights in early layers change and the inputs of later layers vary a lot.
- Each layer must readjust its weights to the varying distribution of every batch of inputs, which slows training.
- Varying input distribution affects the neuron output that enters saturated sigmoid (near zero derivative).
  - The vanishing gradient (i.e. for sigmoid). For sigmoid $\theta(x)$ activation , as $|x|$ increases, $\theta^{'}(x)$ tends to zero.



Remedy

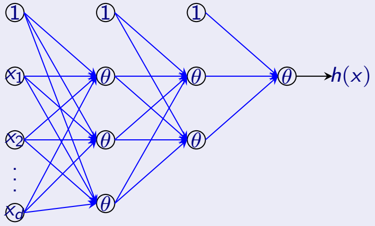
Normalise initial weights and use batch normalisation.

Regularisation

Initialisation: normalise the weight distribution to ensure signal propagation

Assuming:

- The activations are in the linear range $\theta(s) = s$
- The network inputs $x^{(0)}$ are zero mean and i.i.d.
- The weight distribution is i.i.d with zero mean for each layer



Then

$$\mathrm{Var}\left[x^{(l)}\right] = \mathrm{Var}\left[x^{(0)}\right] \prod_{k=1}^{l-1} d^{(k)}\,\mathrm{Var}\left[W^{(k)}\right]$$
$$\mathrm{Var}\left[\delta^{(l)}\right] = \mathrm{Var}\left[\nabla\mathcal{L}\right] \prod_{k=l}^{L-1} d^{(k+1)}\,\mathrm{Var}\left[W^{(k)}\right]$$

$\mathrm{Var}\left[x^{(k)}\right]$ and $\mathrm{Var}\left[W^{(k)}\right]$ is variance in the input and weight matrix in layer k

To preserve the signal through the network in both the forward and backward passes we want:

$$\mathrm{Var}\left[x^{(l)}\right] \sim \mathrm{Var}\left[x^{(l+n)}\right] \Rightarrow d^{(l)}\,\mathrm{Var}\left[W^{(l)}\right] = 1 \;\; \forall d$$
$$\mathrm{Var}\left[\delta^{(l)}\right] \sim \mathrm{Var}\left[\delta^{(l+n)}\right] \Rightarrow d^{(l+1)}\,\mathrm{Var}\left[W^{(l)}\right] = 1 \;\; \forall d$$

A compromise between these two constraints is Xavier initialisation.

Regularisation

- In general practice biases are initialized with 0 and weights are initialized with random numbers sampled from uniform or Gaussian distribution
  - Initializing weights to zeros makes derivative with respect to loss function the same for every w in $W^{(l)}$, thus all weights have the same value in subsequent iterations. This makes hidden units symmetric and continues for all iterations i.e. not better than a linear model.

Xavier initialisation for sigmoid activation

Initialize the weights such that :

$$\mathrm{Var}\left[W^{(l)}\right] = \frac{2}{d^{(l)}+d^{(l+1)}}$$

$$\text{e.g. } W^{(l)} = U\left[-\frac{\sqrt{6}}{\sqrt{d^{(l)}+d^{(l+1)}}}, \frac{\sqrt{6}}{\sqrt{d^{(l)}+d^{(l+1)}}}\right], \quad \mathbf{b} = 0$$

Kaiming He Initialisation for ReLu activations

Initialize the weights such that :

$$\mathrm{Var}\left[W^{(l)}\right] = \frac{2}{d^{(l)}} \ \forall l \neq 1, \text{ with } \mathrm{Var}\left[W^{(1)}\right] = \frac{1}{d^{(1)}} \Leftarrow$$ no RELU on input signal.

Regularisation

Orthogonalisation

The repeated application of the weight matrix W (i) in the forward pass and in the equation

$$\delta^{(l)} = \left(\prod_{k=l}^{L-1} \theta'(s)^{(k)} (W^{(k)})^T\right) \theta'(h(\mathbf{x}))\nabla\mathcal{L}$$

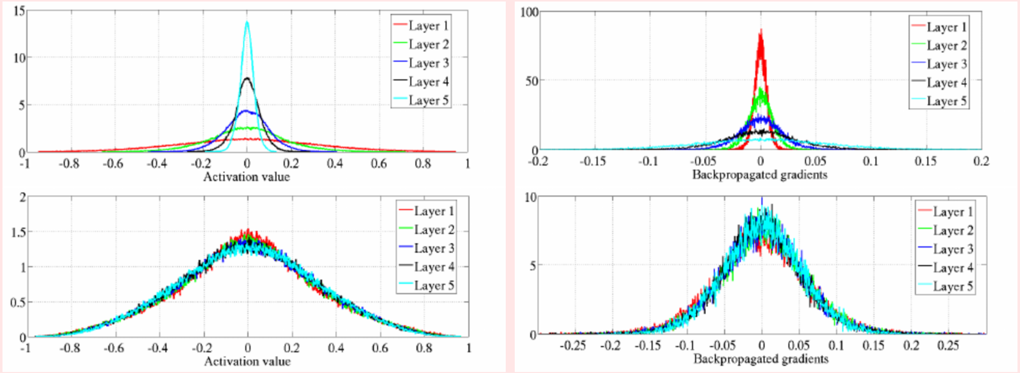
suggests an orthogonal initialisation to prevent the forward and backward signal from vanishing/exploding.

In practice, this can be implemented by randomly initialising a weight matrix $\bar{W}$ and computing the singular value decomposition $\bar{W} = U\Lambda V$ and use matrix V^T as the initial weights

Regularisation

Activation & gradient histograms

Top: Unnormalised initialisation

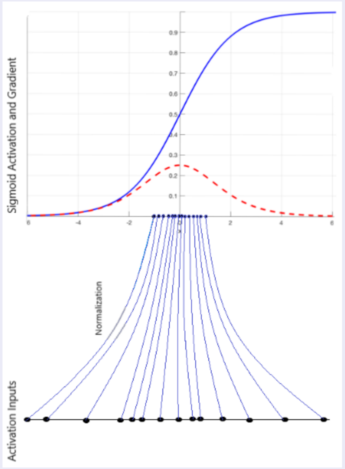


Bottom: Xavier initialisation

Regularisation

Batch Normalisation

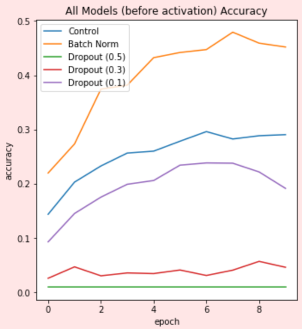
- Making layer inputs similar in distribution allows the network to focus on learning the difference between classes.
- Restricting neuron output to the sweet spot ensures that each layer will pass back a substantial gradient during backpropagation.
  - faster training times, and better performance
- **Training:** batch normalisation transforms layer outputs into a Gaussian distribution, zero centred by batch mean μ_B and unit batch variance by σ_B^2 : $\bar{x} = \frac{x-\mu_B}{\sqrt{\sigma_B^2+\varepsilon}}$, with small ε to avoid 0, followed by rescaling $y = \gamma\bar{x} + \beta = BN_{\gamma,\beta}(\bar{x})$
- **Testing:** normalise by the average and variance of the training population i.e. averages of all batch means and variances calculated during training.
- Since the output of one layer is the input of the next, layer inputs will also have significantly less variation from batch to batch.
- BN layer is inserted between convolution and activation layers

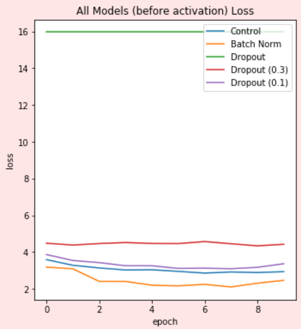


Regularisation

Batch Normalisation Effect

- Ideally we would like to whiten each layer's activations, but this is too expensive
  - Instead, normalize each feature independently in each layer
  - Use minibatch statistics to approximate the statistics of the training set
- Prevents vanishing gradient in networks with saturable nonlinearities (sigmoid, tanh, etc)
- Regularizing effect
- Allows for higher learning rates thus decreasing training time
- Resulting in better performance
- Additional hyperparameters that can be tweaked

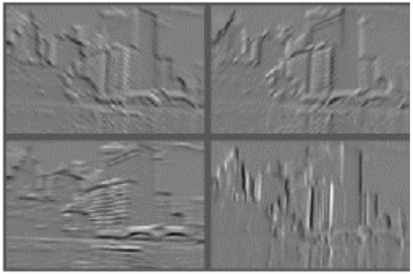




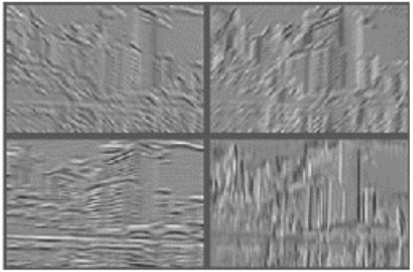
Practical hints

- Batch normalisation of each feature map
  - e.g. input data in a 2D ConvNet has shape $[N, H, W, C]$, with
    - ★ N is the number of examples in the minibatch
    - ★ $H \times W$ are image height and width
    - ★ C is the number of channels
  - Standard BN would compute $H \times W \times C$ means and stddevs to normalise each feature separately at each spatial location
  - BN in ConvNets instead computes C means and stddevs and normalises jointly for all locations (feature map)
    - ★ to respect the structure (spatial patterns)
  - Enhanced details with BN

Activation map without BN



Activation map with BN



Summary

- Backpropagation (reminder)
  - Vanishing and exploding gradients
- Optimizers
  - Nesterov, Adagrad, RMSProp, Adadelta, Adam
- Regularisation
  - Dropout, initialisation, batch normalisation