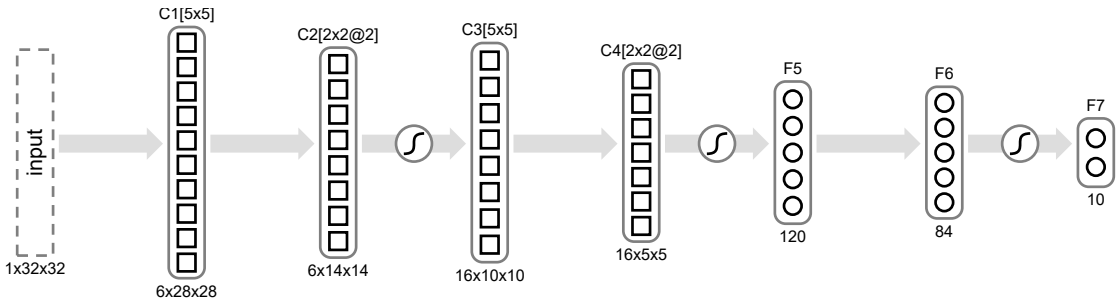
University of Cambridge
Engineering Part IB
Information Engineering Elective

Paper 8: Image Searching and Modelling Using Machine Learning

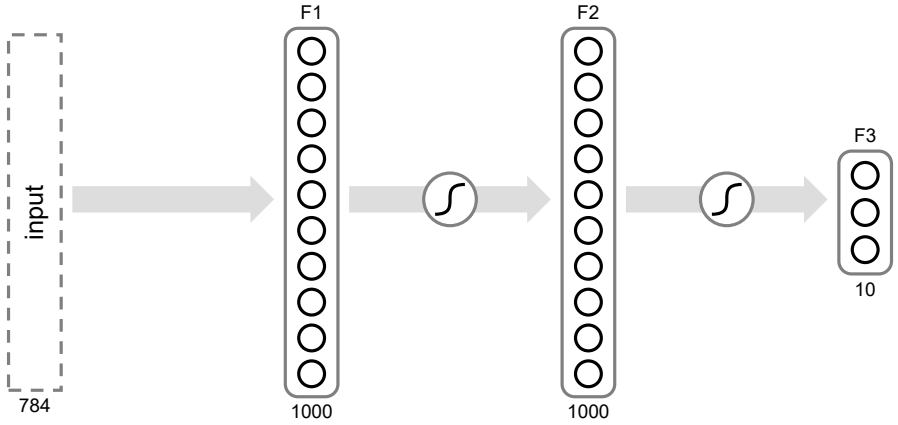
Handout 3: Convolutional Neural Networks



Roberto Cipolla and Matthew Johnson
May 2017

Problems of Scale

Near the end of the previous lecture, we looked at the following multi-layer perceptron:



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

$$|\mathbf{W}_0| = 784 \times 1000 = 784000$$
$$|\mathbf{W}_1| = 1000 \times 1000 = 1000000$$
$$|\mathbf{W}_2| = 1000 \times 10 = 10000$$

This is quite a few more than the networks we originally explored. The number of parameters is not just an issue for the computation, however. Every parameter in the network increases its ability to store information, essentially providing it with more memory. If a network has too much capacity, it can begin to memorize data points and bypass generalization. It is fairly urgent that we find a way to reduce the number of parameters in these deep networks for computer vision.

SGD Tricks

Before we dive into ways to reduce the number of parameters in deeper networks, however, there are more fundamental problems to overcome. So-called "vanilla" stochastic gradient descent can have problems making headway with so many parameters, but there are some tricks we can use that help.

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:

$$\nabla \mathbf{W}^{\tau+1} = \frac{\partial L}{\partial \mathbf{W}^{\tau}} + \epsilon \nabla \mathbf{W}^{\tau}$$
$$\mathbf{W}^{\tau+1} = \mathbf{W}^{\tau} - \eta \nabla \mathbf{W}^{\tau+1}$$

at training step τ where ϵ is the momentum hyperparameter and η is the learning rate.

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 η^0 is multiplied by γ every σ steps.

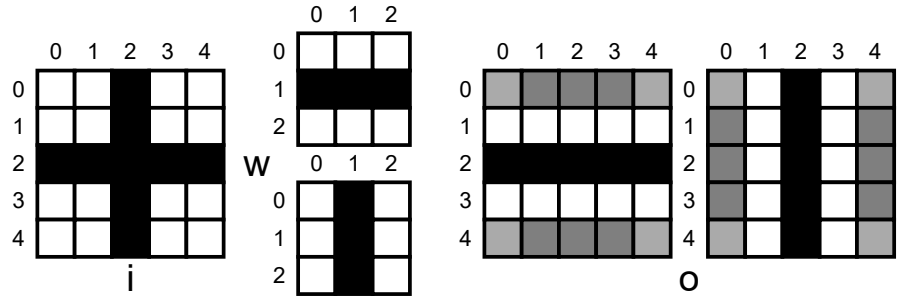
Inverse Learning Rate Updates Another method is to continuously alter the learning rate in inverse proportion to the number of steps:

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

where γ and ρ control the speed of learning rate decay.

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:

$$\begin{aligned} o_{f,r,c} = {} & w_{f00} i_{r-1,c-1} + w_{f01} i_{r-1,c} + w_{f02} i_{r-1,c+1} \\ & + w_{f10} i_{r,c-1} + w_{f11} i_{r,c} + w_{f12} i_{r,c+1} \\ & + w_{f20} i_{r+1,c-1} + w_{f21} i_{r+1,c} + w_{f22} i_{r+1,c+1} \end{aligned}$$

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.

Convolutional Layers, cont.

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 $\mathbf{x}$. We extract patches in row column order and embed them as columns in a matrix, as shown below:

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

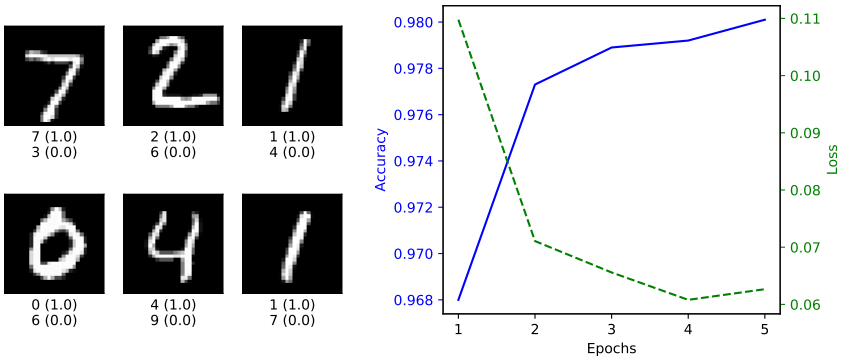
We will denote this embedding operation as $\mathbf{X} = \mathcal{E}(\mathbf{x})$. We can then store the weights in a weight matrix and use matrix multiplication to perform the convolution:

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

$$C_i(x) = \mathbf{W}_i \mathbf{X}$$

CNN on MNIST

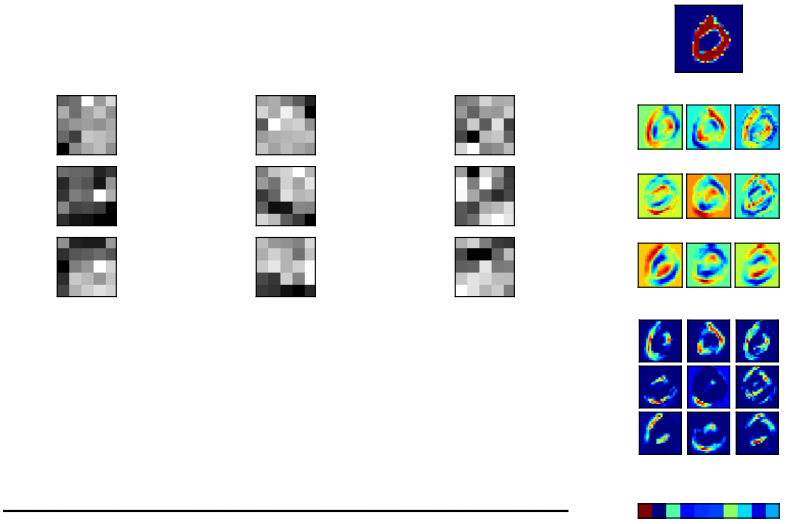
Let us see how this CNN performs on our MNIST task:



This is the best system yet, certainly, but there is a worrying thing happening in the validation loss: it is starting to go up. This is because the final layer is a 5184×10 matrix, which means that this network has many more hidden units than previous ones we have analyzed. If we had 5184 hidden units in a traditional Multi-Layer Perceptron it would require $784 \times 5184 = 4,064,256$ parameters, but the convolutional layer only has $9 \times 5 \times 5 = 225$ weights. That is good, as it keeps us from over-parameterizing the model. However, the 9 filters create 9 images, which have $9 \times 24 \times 24 = 5184$ pixels, and thus we need 51,840 parameters to reduce those down to the output space of 10 values. Even though we have drastically reduced the number of parameters, the validation loss going up probably means the model is overfitting to the data. We need to reduce that number further.

CNN on MNIST, cont.

Before we worry about the overfitting, let us look at what the system has learned:

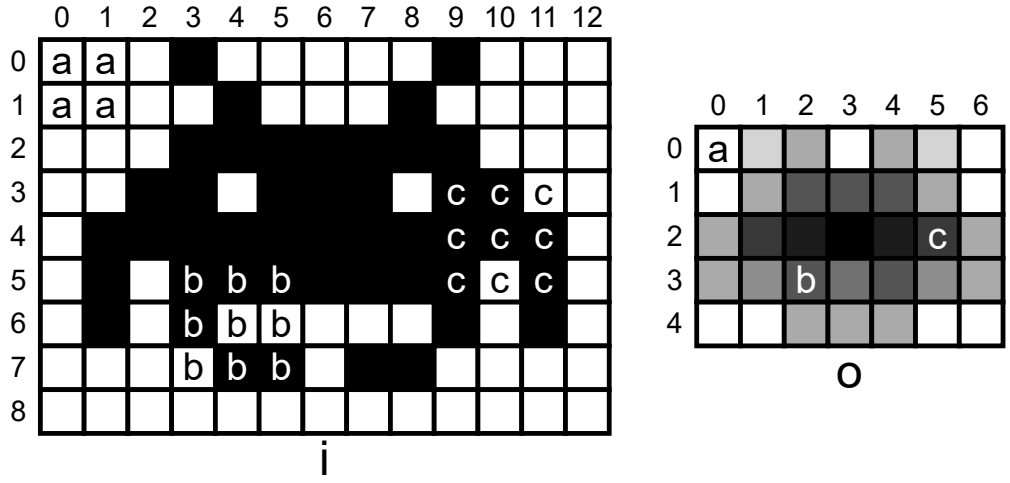


As can be seen, they are both familiar and yet very different to the filters that we used earlier. We can tell from the output images what kinds of features the filters are detecting, but they are also doing more subtle things, which you can see in the images after a ReLU has been applied.

That said, we do need to somehow reduce the number of pixels that are output by our convolutional layer. For this, we shall use a technique called pooling.

Average Pooling

The concept of pooling is similar to sub-sampling, whereby an image is reduced in size by representing a region in the input image via a single pixel. In standard sub-sampling this value is a single pixel in the input region, *e.g.* the center. In average pooling, the value is the mean value of the pixels in the input region.



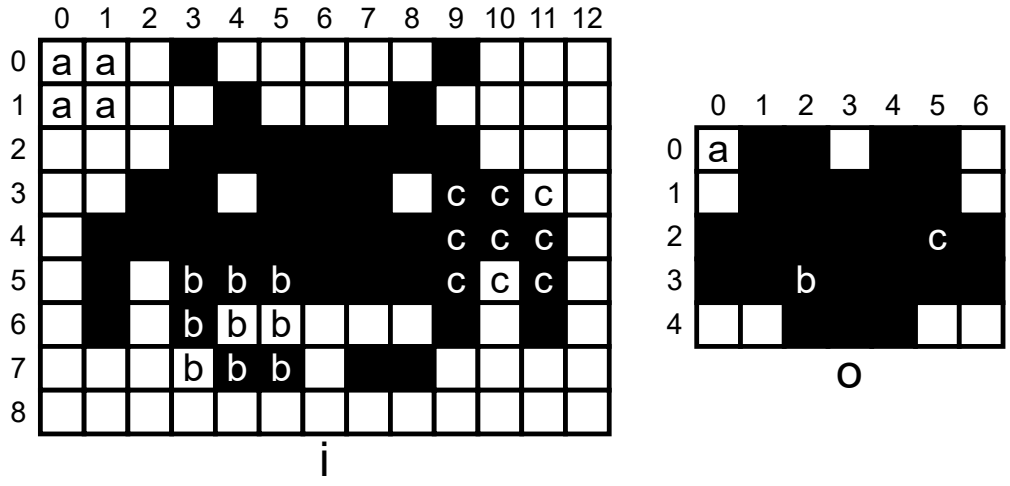
As can be seen above a filter of size 3 is applied at a *stride* of 2, though the size and stride can be altered in practice as needed by the problem. This pooling acts as a *non-parameterised* layer in the neural network, much like non-linearities. An output value is computed as follows:

$$o[r, c] = \frac{1}{9} \sum_{k=0}^{2} \sum_{l=0}^{2} i[2r + k - 1, 2c + l - 1]$$

This is equivalent to using the embedding function $\mathcal{E}$ and then multiplying by a weight matrix $\mathbf{W} = \left\{ W_{i,j} | \forall i \forall j \; W_{i,j} = \frac{1}{9} \right\}$. When treated in this way the derivative is also this matrix (as in this case $\mathbf{W}^{\mathrm{R}} \equiv \mathbf{W}$).

Max Pooling

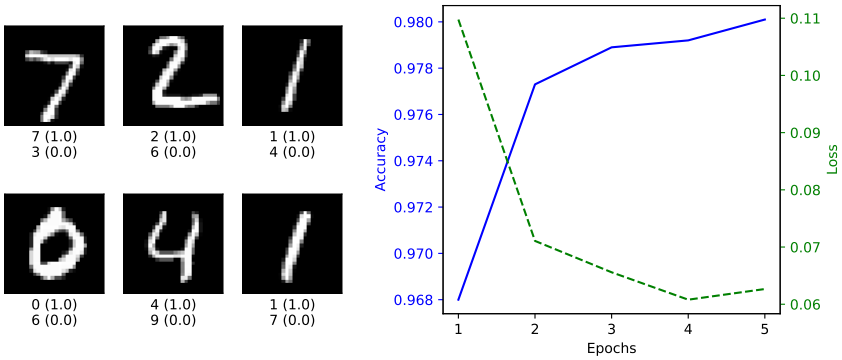
An alternative to average pooling is maximum, or max, pooling. As with average pooling, the max pooling operator has a set size and is applied at a predetermined stride. However, instead of taking the arithmetic mean of all of the pixel values it selects only the maximum value as its output value.



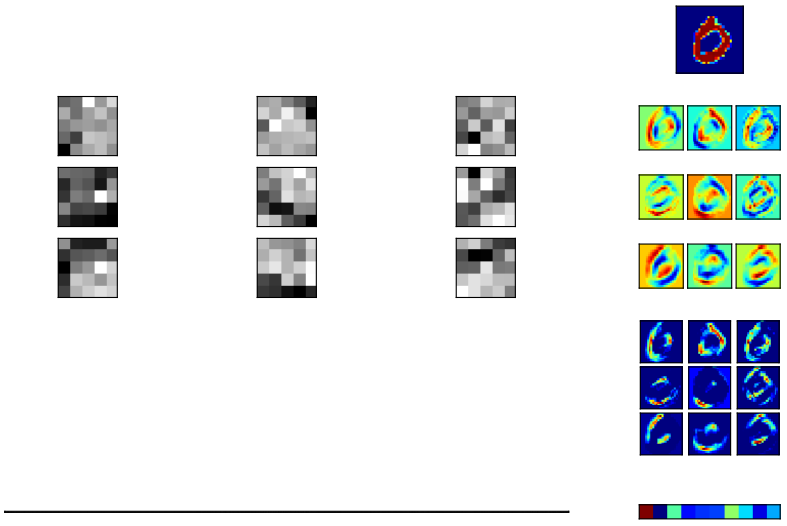
This creates a slight difficulty during backpropagation, as the partial derivatives passing backwards through the operator only apply to a single input pixel. We can achieve this by creating, during each forward pass, a $O \times I$ matrix $\mathbf{W}$ in which each row has a single value of 1 at the column i corresponding to the maximum input value used for the output row o. The derivative is then $\mathbf{W}^{\mathrm{T}}$.

CNN + Pooling on MNIST

When we run the training with a new CNN that incorporates a max pooling layer, we see that while it uses far fewer parameters this does not reduce its accuracy:



You can see below how the pooling layer makes just particular features stand out. As you chain together convolutional layers and max pooling layers and the dimensionality of the images decreases, more and more high-level features of the image can be detected and used for classification by the final layer.



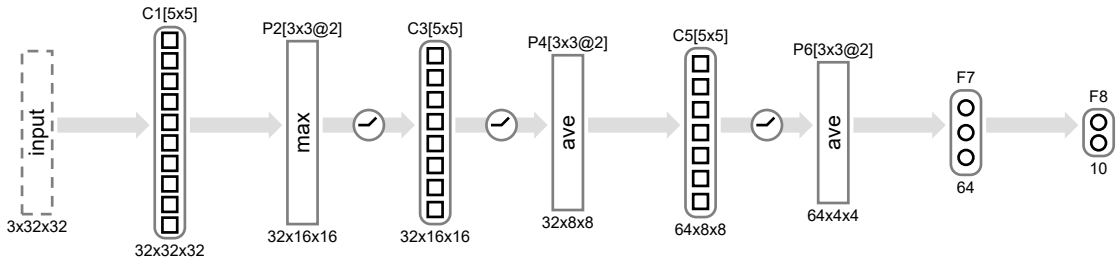
CIFAR-10

We have seen some pretty impressive things so far with the artificial neural nets we have studied, but it is time to make things much harder. CIFAR-10, while it has many of the same characteristics as MNIST, is a considerable step up in difficulty. It is drawn from the "80 million tiny images" dataset [5] which consists of (almost) 80 million 32×32 RGB images downloaded from the internet and labeled with one of 75,062 non-abstract nouns in English.

CIFAR-10 (named after the Canadian Institute for Advanced Research where it was gathered) consists of 60,000 images gathered from this larger set by gathered by Alex Krizhevsky, Vinod Nair and Geoffrey Hinton [3]. These images belong to 10 classes: (in order from left to right above) airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. As can be seen, the dataset contains a lot of intra-class variance, in addition to complications brought about by color and the much larger variety inherent in non-curated natural images.

Anatomy of a Deep Net

Using these new building blocks, we can now fully examine the anatomy of a modern deep net for CIFAR-10.



We can compute the number of parameters per trainable layer: Channels $\times$ Size^2 $\times$ Nodes.

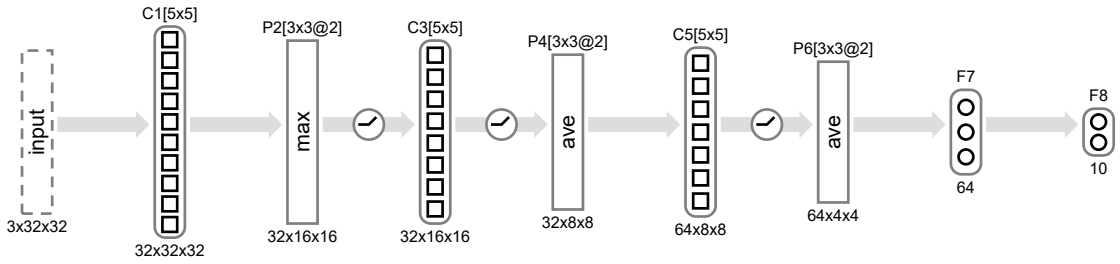
Name	Channels	Size	Nodes	# Parameters
C1	3	5	32	2400
C3	32	5	32	25,600
C5	32	5	64	51,200
F7	1024		64	65,536
F8	64		10	640
Total				145,376

We can also calculate the input and output dimensions. For convolutional and pooling layers, output is a function of stride, size and padding: $dim_o = \lfloor \frac{dim_i + 2 \times \text{padding} - \text{size}}{\text{stride}} \rfloor + 1$.

Name	Input	Size	Stride	Pad	Nodes	Output
C1	3x32x32	5	1	2	32	32x32x32
P2	32x32x32	3	2	1		32x16x16
C3	32x16x16	5	1	2	32	32x16x16
P4	32x16x16	3	2	1		32x8x8
C5	32x8x8	5	1	2	64	64x8x8
P6	64x8x8	3	2	1		64x4x4
F7	1024				64	64
F8	64				10	10

Anatomy, cont.

Now that we understand the parameters and the dimensionality, let us look at this in more detail.



Each layer in the network plays an important role.

C1 Extracts low-level features in the image, like edges, corners and blobs.

P2 Provides some flexibility of location to the low-level features

C3 Looks for parts that are combinations of lower-level features

P4 Smoothes the part responses before subsampling

C5 Finds structures that are built from parts

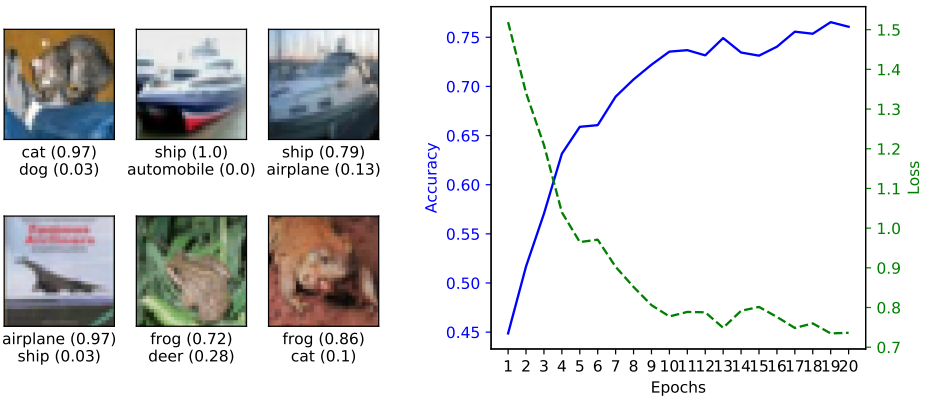
P6 Smoothes and subsamples the structural responses. The output of this final layer acts as a CIFAR-10 specific feature vector of length 1024.

F7 Sub-category classifiers

F8 Final classifier

Deep CNN on CIFAR-10

Now that we have designed a DNN for CIFAR-10, let us see how it does:



It does rather well, though it still is uncertain about some of its labels. The confusions, however, make sense: cat then dog in the top left, airplane or ship. Frog or deer seems counterintuitive, but upon examination many deer images have this color pattern of brown on green. Look at the loss curve. This network was trained for 20 epochs, and we can see several points when the loss went up before going back down again. This behavior arises as the network, by virtue of stochastic gradient descent, leaves a local minimum in the objective. These jumps are to be expected on difficult tasks like this one, and indicate the networks ability to continue learning as it sees more and more data.

Deep Net Training

The following is a typical procedure for training a deep neural network on a dataset.

1. Divide the dataset into *training* $\mathbf{D}$, *validation* $\mathbf{V}$ and *test* $\mathbf{T}$ partitions
2. Compute the "mean" image of $\mathbf{D}$, and subtract it from all images
3. Scale all images so that pixel values are in the range $[-1, 1]$
4. Design a network architecture, or adapt a known-good architecture to the dataset
5. Select an optimisation algorithm (*i.e.* a version of SGD)
6. For a pre-determined set of epochs, do the following:
   (a) Randomly sample (without replacement) $\mathbf{D}_B$ images from $\mathbf{D}$
   (b) Compute $\nabla \mathbf{W}^\tau$ for all $\mathbf{W}$ in the model
   (c) Update all $\mathbf{W}^\tau \rightarrow \mathbf{W}^{\tau+1}$
   (d) Once all images in $\mathbf{D}$ have been seen (*i.e.* when an epoch is complete), evaluate the network on $\mathbf{D}$ and $\mathbf{V}$ to monitor changes in the loss
7. Evaluate final performance on $\mathbf{V}$, and repeat the above with different optimisation hyperparameters as necessary
8. Retrain the network using $\mathbf{D} \cup \mathbf{V}$ with the final hyper-parameters, and evaluate on $\mathbf{T}$

Data Augmentation

When it comes to deep networks, the only thing better than a lot of data is a whole lot of data. However, gathering supervised data is very expensive. One way to overcome this problem is via *data augmentation*, whereby on each epoch each training image is turned into several additional images by way of translated cropping and horizontal reflection:

In this way one training image can become 10. Another common usage of data augmentation is during the testing phase, whereby instead of simply computing $P(\mathbf{x} = i)$ from a single image, $\mathbf{x}$ is altered as shown above and then each version of $\mathbf{x}$ is shown to the network. In this scenario, the predicted label is then:

$$i = \underset{j}{\operatorname{argmax}} \left\{ j_a | \forall a : j_a = \underset{k}{\operatorname{argmax}} P(\mathbf{x}_a = k) \right\}$$

This can result in significant improvement regardless of the network model used.

Batch Normalisation

Batch Normalisation [2] is a technique that is essential to the deepest neural net architectures, but also useful in general for stabilising and accelerating the training process in all circumstances. The concept is straightforward: if we knew the statistics of each layer's output over the entire dataset, we could normalise those outputs so that all output vectors had zero mean and unit variance. In mathematical terms, we want to do the following:

$$\mu_{\mathbf{D}} = \frac{1}{|D|} \sum_{\mathbf{x} \in \mathbf{D}} \mathbf{x}$$

$$\sigma^2_{\mathbf{D}} = \frac{1}{|\mathbf{D}|} \sum_{\mathbf{x} \in \mathbf{D}} (\mathbf{x} - \mu_{\mathbf{D}})^2$$

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - \mu_{\mathbf{D}}}{\sqrt{\sigma^2_{\mathbf{D}} + \epsilon}}$$

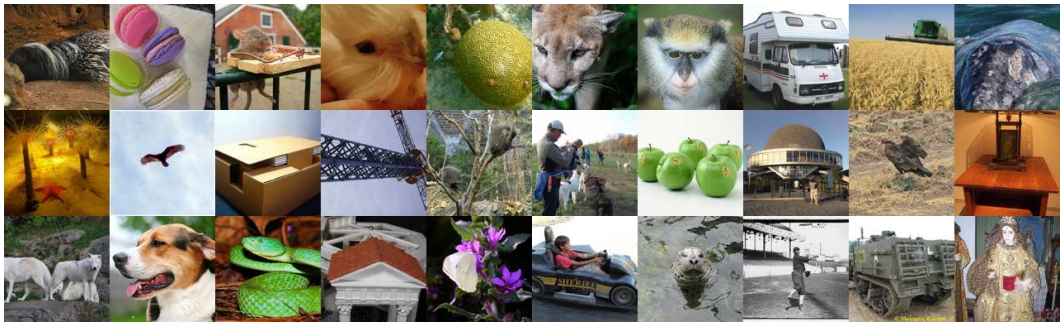
The problem is that we do not know $\mu_{\mathbf{D}}$ or $\sigma^2_{\mathbf{D}}$, nor can we compute them as they will change as the network itself changes during training. We overcome this by using the same batch trick to estimate these values *i.e.* by computing them for each batch during training. Once the network is trained, we can then compute the true values and use those during testing. We can even learn training parameters for scaling and shifting the normalised vectors:

$$\mathbf{y} = \gamma \hat{\mathbf{x}} + \beta$$

Backpropagation through this operation is outside of the scope of these lectures, but is detailed in the referenced paper.

ImageNet

The final dataset we will look at is the ImageNet dataset. It consists of over 14 million images mapped to around 20,000 labels. The images are extremely varied, coming from a variety of different sources all over the internet, and have each been hand-labeled by a human.



Every year since 2010 there has been a challenge organised around subsets of these images. In 2015, it was dominated by Microsoft Research Asia [4].

Detection The algorithm has to specify for each image multiple classes (from a choice of 200) and their locations in the image.

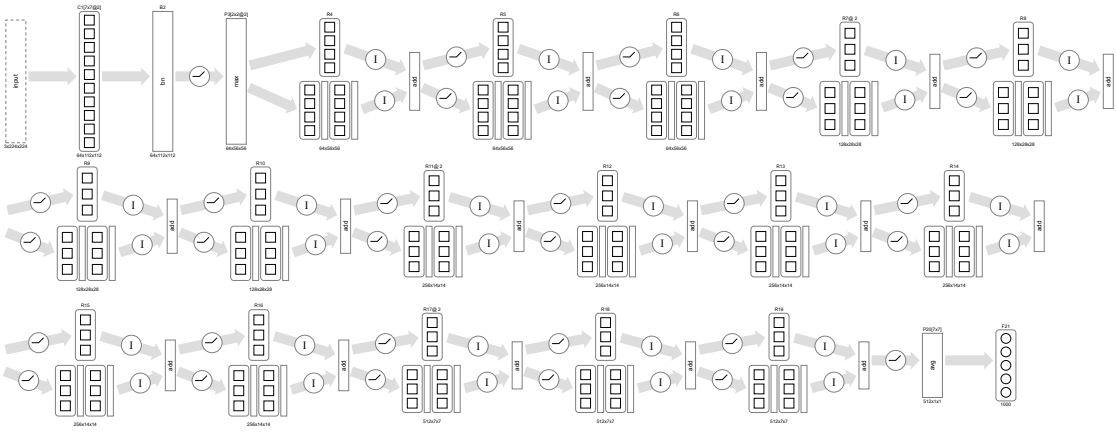
Team Name	# Categories	Mean AP
MSRA	**194**	**0.621**
Qualcomm Research	4	0.536
CUImage	2	0.527

Classification The algorithm has to specify for each image to which class it belongs (from a choice of 1000)

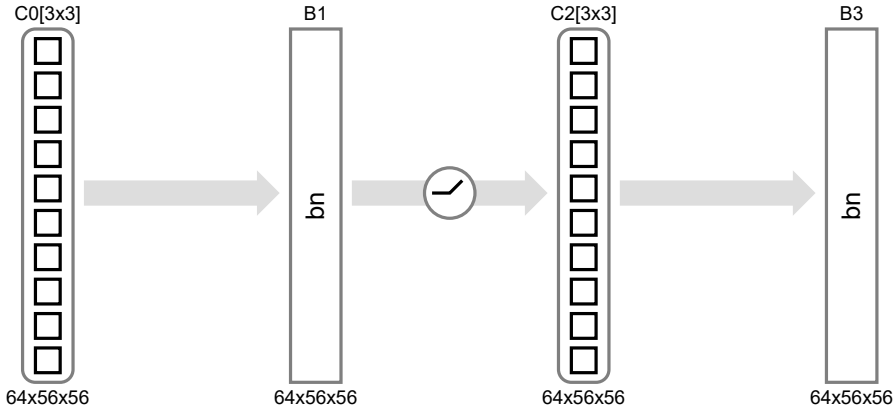
Team Name	Error
MSRA	**0.0357**
ReCeption	0.0358
Trips-Soushen	0.0458

ResNet

ResNet [1] used modular construction to assemble the deepest successfully trained neural network to date. The overall structure of the network (in a 34-layer configuration) looks like this:



The bottom of each R block in that diagram is a neural net in miniature of the following form:



Note that while ResNet can be extrememly deep (with up to 152 layers) the building blocks are the elements we have already seen in these lectures. The essential techniques that make the training of this network possible are batch normalisation and the shortcut structures, which enable the network to selectively exclude blocks during training and then add them in as they become useful.

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[2] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the* 32^{nd} *International Conference on Machine Learning*, Lille, France, 2015.

[3] Alex Krizhevsky. Learning multiple layers of features from tiny images. Master's thesis, University of Toronto, 2009.

[4] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[5] Antonio Torralba, Robert Fergus, and W. T. Freeman. 80 million tiny images: a large dataset for non-parametric object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(11):1958–1970, 2008.