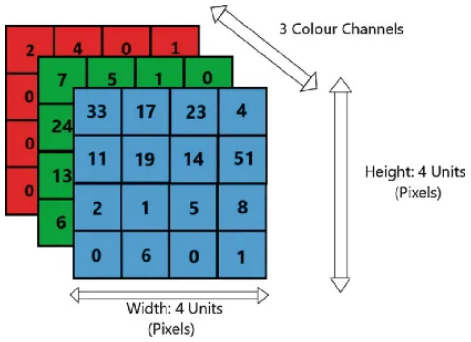
EMAT31530: Convolutional neural networks

Laurence Aitchison

So far, we've seen lots of material on how to train NNs using backprop, and how to use PyTorch. But to actually apply NNs to e.g. image datasets (for instance for classifying the objects in the image), we need to consider architectures specifically adapted to images. Perhaps the key operation is "convolution", which is the basis for all the classic and many of the modern networks for images. Networks that involve convolutional layers are called "convolutional neural networks", and that's what we'll look at this week!

1 Images and feature maps as tensors

To understand convolutions, the first step is to understand how we represent images in deep learning. We represent images as big 3D tensors, with shape $3 \times \text{height} \times \text{width}$,



The three colour channels represent the red, green and blue channels in the image (i.e. the amount of red, green or blue at that pixel). Alternatively, a greyscale image is a $1 \times \text{height} \times \text{width}$ image.

A feature map is very similar to an image. The key difference is that a feature map has much larger number of channels (usually around $100 - 500$), rather than 3 in a colour image or 1 in a greyscale image. So a feature map is of shape $\text{channels} \times \text{height} \times \text{width}$, which is usually abbreviated to $C \times H \times W$. You can imagine that in a feature map, we have assigned a feature vector to every pixel in every image. That contrasts to fully-connected networks, where we'd just have one feature vector associated with every image.

In practice, we usually work with a minibatch of N images. Thus, we end up working with tensors of size minibatch size $\times$ channels $\times$ height $\times$ width, which is usually abbreviated to $N \times C \times H \times W$.

Convolutional layers take images/feature maps as inputs, and returns images/feature maps as outputs. We'll see how they operate in the next section.

2 Convolutions

To start, we draw a diagram for the simplest possible deep-learning style convolution. Specifically, consider a convolution with one input channel and one ouput channel, and with convolutional weights (or "kernel"),

These weights are 3×3, so we say that the "kernel size" is 3.

We slide these weights over the input image/feature map. At each location, we calculate the product between the weight and the image pixel. We sum all these products to get the output in the current location.

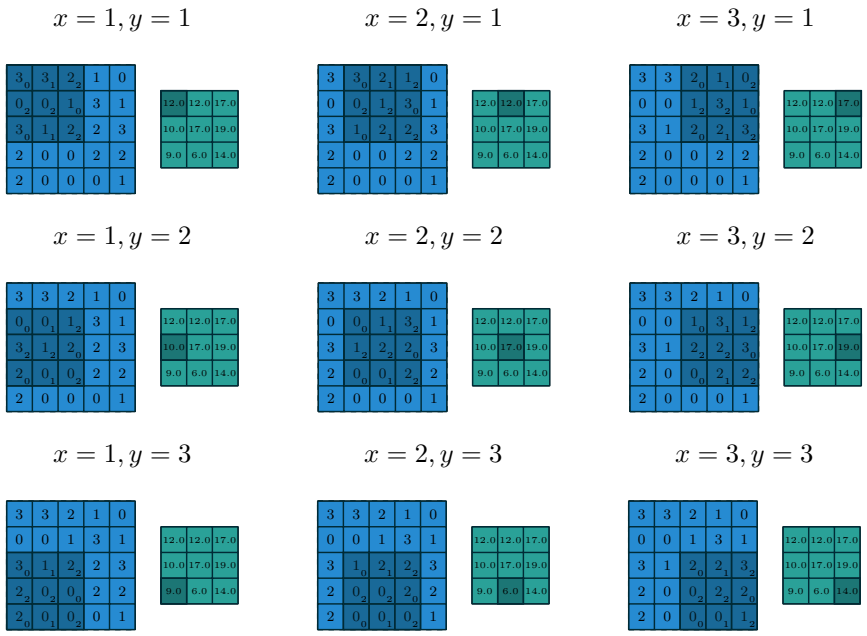


Figure 1: Computing the output of a simple convolutional layer. The blue 5×5 square shows the inputs (h in Eq. 1), the green 3×3 square shows the output ($a_{x,y}$ in Eq. 1). The darker 3×3 square in the input shows the current location of the patch of weights as it slides over the image. This diagram along with others in this pdf are taken from "A guide to convolution arithmetic for deep learning", by Vincent Dumoulin and Francesco Visin. This is a great article to check out if you're interested in filling in any gaps.

Specifically, the formula for this convolution is,

$$a_{x,y} = \sum_{\delta_x=-1}^{1} \sum_{\delta_y=-1}^{1} h_{x+\delta_x,y+\delta_y} W_{\delta_x,\delta_y} \tag{1}$$

here:

- $a_{x,y}$ is the output image / feature map (in green on the right in Fig. 1). In this example, the outputs are size 3×3. So the indices x and y represent location in the output image, and range from 1 to 3. As an example, in Fig. 1, $a_{1,1} = 12.0$ (the indexing starts in the top-left).
- h is the input image / feature map (in blue on the left in Fig. 1). In this example, the inputs are size 5×5, and the indices range from 0 to 4. As an example, in Fig. 1, $h_{0,0} = 3$.
- The indices $x+\delta_x$ and $y+\delta_y$ represent the location within the darker blue patch location in the inputs, and range from 0 to 4.
- W_{δ_x,δ_y} is the convolutional weights (in grey in the first image, or in the 3×3 darker patches in the inputs in Fig. 1). In this example, the weights are size 3×3. So the indicies δ_x and δ_y represent the location within the 3×3 weight matrix, and range from -1 to 1. As an example, looking at the previous diagram, and starting our indexing in the top-left, $W_{-1,-1} = 0$.

Note that here, when we say that "an index ranges from -1 to 1", that's for the purpose of writing down these equations. You can't change how Python/PyTorch indexes tensors (so e.g. in Python/PyTorch, indices always start from 0).

2.1 Convolutions with multiple input and output channels

In the previous section, we considered the simplest setting for convolutions, with a single input channel, and a single output channel. However, in practice in deep learning, we almost always have multiple input and multiple output channels. What happens in that setting? Well, it is almost exactly the same as in the single-input and single-output channel settings, except:

- Instead of an scalar at each location in the input and output (i.e. $h_{x+\delta x,y+\delta y}$ and $a_{x,y}$), there's a whole feature *vector* at each location, $\mathbf{h}_{x+\delta_x,y+\delta_y}$ and $\mathbf{a}_{x,y}$.
- Instead of just a scalar weight at each location in the patch (i.e. W_{δ_x,δ_y}) that there's a whole weight matrix, $\mathbf{W}_{\delta_x,\delta_y}$.

Overall, that means a 2D convolution, with multiple input and output channels can be written,

$$\mathbf{a}_{x,y} = \sum_{\delta_x=-1}^{1} \sum_{\delta_y=-1}^{1} \mathbf{h}_{x+\delta_x,y+\delta_y} \mathbf{W}_{\delta_x,\delta_y}. \tag{2}$$

Here,

- $\mathbf{a}_{x,y}$ is a row-vector of length C_{out} (or `out_channels` in PyTorch). There is one vector at each spatial location in the output.
- $\mathbf{h}_{x+\delta_x,y+\delta_y}$ is a row-vector of length C_{in} (or `in_channels` in PyTorch). There is one vector at each spatial location in the input.
- $\mathbf{W}_{\delta_x,\delta_y}$ is a matrix of size $C_{\text{in}} \times C_{\text{out}}$. There is one of these matrices at each location in the patch.

2.2 Shapes in PyTorch in practice

Finally, it is worth mentioning the shape of the inputs and weights in actual convolutions in PyTorch.

- The inputs and outputs are standard feature maps, of size $N \times C_{\text{in}} \times H_{\text{in}} \times W_{\text{in}}$ and $N \times C_{\text{out}} \times H_{\text{out}} \times W_{\text{out}}$. Note that the inputs and outputs can be of different sizes (e.g. in Fig. 1 we had $5 = H_{\text{in}} = W_{\text{in}}$ and $3 = H_{\text{out}} = W_{\text{out}}$). Finally, N is the minibatch size.
- The weights are of size $C_{\text{out}} \times C_{\text{in}} \times H_{\text{kernel}} \times W_{\text{kernel}}$, where H_{kernel} and W_{kernel} are the size of the convolutional patch (or "kernel"); (in Fig. 1 we had $3 = H_{\text{kernel}} = W_{\text{kernel}}$).

2.3 Convolutions in (Engineering) Maths [Non-Examinable]

If you have not encountered convolutions before, you can ignore this section. However, if you have encountered convolutions before in an (Engineering) maths context, you might be slightly confused, as convolutions are typically defined as,

$$a(x) = \int_{-\infty}^{\infty} d\delta' \; h(x - \delta') \; w(\delta') \tag{3}$$

Note that here, δ is a variable, not the Kronecker/Dirac delta. This is an operator that takes two functions (h and w) as input, and returns a function, a as output. This doesn't look much like the AI-version of convolution. We can make it look at bit more similar by setting, $\delta = -\delta'$

$$a(x) = \int_{-\infty}^{\infty} d\delta \; h(x + \delta) \; w(-\delta). \tag{4}$$

If we write down a discrete version of this expression, we get something that looks alot like an AI-style convolution, but in 1 dimension,

$$a_x = \sum_{\delta=-\infty}^{\infty} h_{x+\delta} W_{-\delta}. \tag{5}$$

There are two key differences:

- Here, δ ranges from $-\infty$ to ∞. In contrast, in AI (Eq. 1), δ has a small range, e.g. often from -1 to 1.
- Here, there's a $-\delta$ in $W_{-\delta}$, while in AI (Eq. 1) there's no minus sign so we just have W_δ. It turns out that this doesn't matter in AI, as all the elements of the weight are learned.

2.4 Convolutional Strides and Padding

When practically using convolutions in deep learning, there are a few options to set. In particular, the "kernel size", "strides" and "padding". Previously, we have used the simplest (but perhaps not most standard) setting of stride=1 and padding=0.

Kernel size is the spatial size of the patch of weights that we slide over the image.

Strides give the size of the "jumps" as we slide the weights over the input image: see examples below. As such, $H_{\text{out}} \approx H_{\text{in}}/\text{stride}$ and $W_{\text{out}} \approx W_{\text{in}}/\text{stride}$.

Padding is a region around the input image that we fill with zeros: see dashed border around the inputs in the examples below. Without padding, the outputs will tend to be smaller than the inputs, even with stride=1. That happened above in Fig. 1.

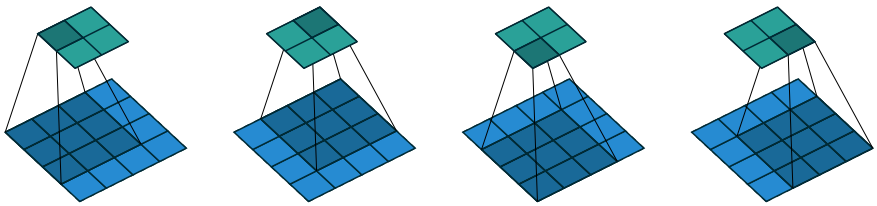


Figure 2: Convolving a 4×4 input using kernel size=3, stride=1 and padding=0.

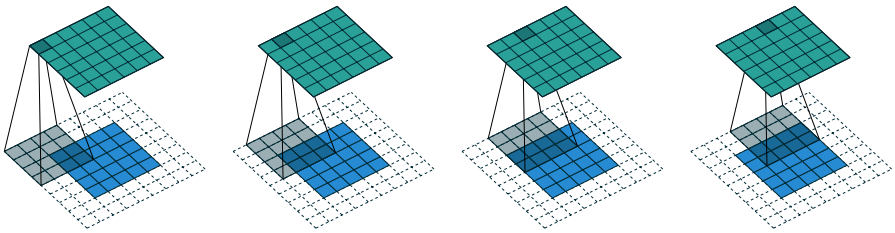


Figure 3: Convolving a 5×5 input using kernel size=4, stride=1 and padding=2.

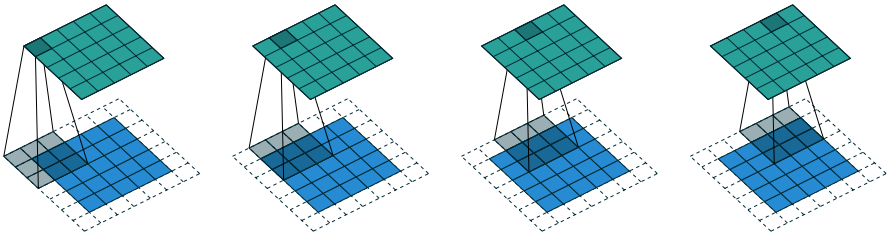


Figure 4: Convolving a 5×5 input using kernel size=3, stride=1 and padding=1. Note this is known as "same" padding, as the input is the same shape as the output.

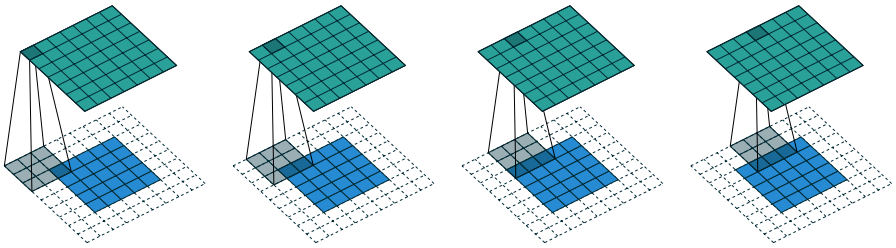


Figure 5: Convolving a 5×5 input using kernel size=3, stride=1 and padding=2.

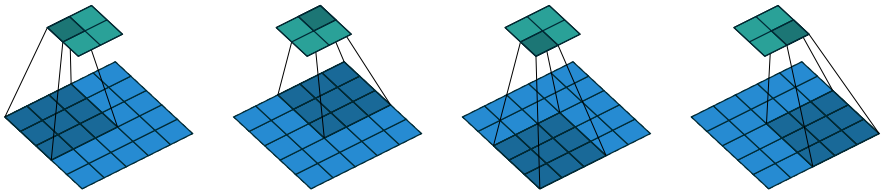


Figure 6: Convolving a 5×5 input using kernel size=3, stride=2 and padding=0.

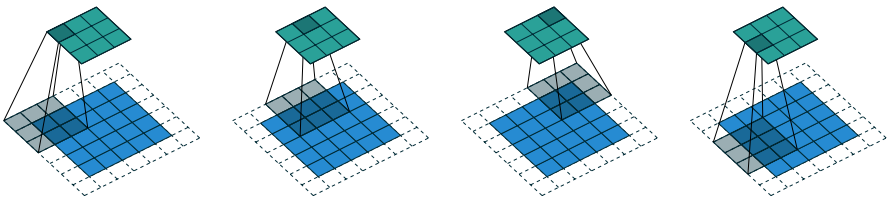


Figure 7: Convolving 3×3 weights over a 5×5 input with stride=2 and padding=1.

3 Pooling

In addition to discrete convolutions themselves, *pooling* operations are another important building block in CNNs. Pooling operations usually reduce the size of feature maps by using some function to summarize subregions, such as taking the average or the maximum value.

Pooling slides a window across the input, just like a convolution. The difference is that instead of multiplying the features in that window by weights, pooling layers feed the features in that layer into a simple *pooling function*. The pooling function is usually max or average. Again, pooling operations have strides and padding that behave in the same way as a standard convolution. Usually, when pooling we use a stride of at least 2, to ensure that we are actually reducing the size of the feature map (though in the example below we use stride=1).

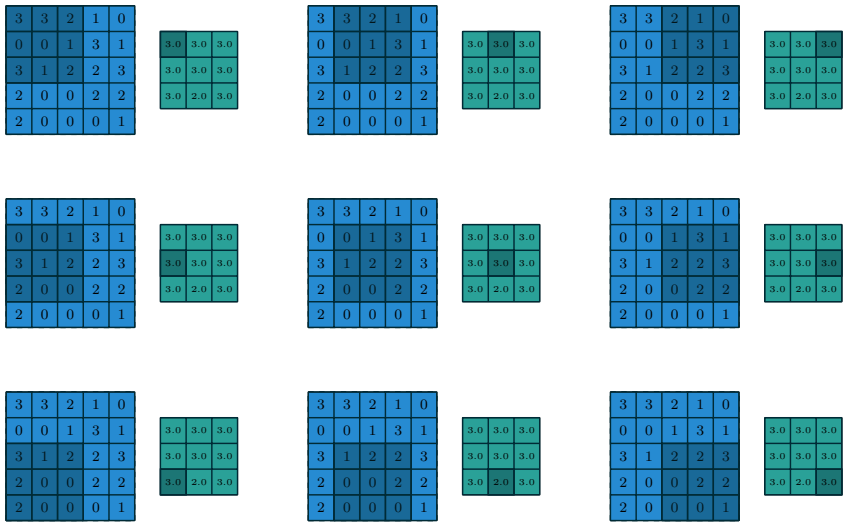


Figure 8: Max pooling on a 5×5 input using kernel size=3, stride=1, padding=0

3.1 Global average pooling

For image classification, we ultimately want just vector of logits as our output (for a single image). But convolutional layers are going to give us a whole feature map of size channels × height × width. So how can we convert a feature map into just a vector? Perhaps the most common method is "global average pooling", which just averages over the spatial dimensions.

$$\text{feature vector}_c = \frac{1}{HW} \sum_{x=1}^{W} \sum_{y=1}^{H} \text{feature map}_{c,x,y} \tag{6}$$

In PyTorch, it is easiest to implement this using `nn.AdaptiveAvgPool2d(1)`, which returns a feature map with height=1 and width=1 (that's what the 1 specifies: if you did `nn.AdaptiveAvgPool2d(3)`, it would return a feature map with height=3 and width=3).

4 Data Augmentation

Often I use convolutional neural networks for classification: i.e. for classifying what object is present in the image. Interestingly, it turns out that if I take an image of a kitten, and:

- Horizontal flip.
- Small rotation.
- Small translation/crop.

I still end up with an image of a kitten.

So I can effectively enlarge my dataset by flipping/rotating/cropping all my images. And this turns out to be a *huge* factor in the success of modern networks. Basically all modern networks for images use some form of data augmentation.

5 Batchnorm

Batchnorm ensures that each channel has a specified, learned mean and variance. First, batchnorm computes the mean-and variance across channels, averaging across images in the minibatch, and across spatial locations.

$$\mu_c = \frac{1}{NHW} \sum_{i=1}^{N} \sum_{x=1}^{W} \sum_{y=1}^{H} \text{input}_{i,c,x,y}, \tag{7}$$

$$\sigma_c^2 = \frac{1}{NHW} \sum_{i=1}^{N} \sum_{x=1}^{W} \sum_{y=1}^{H} (\text{input}_{i,c,x,y} - \mu_c)^2. \tag{8}$$

Then, batchnorm "standardises" the inputs: subtracting the per-channel mean, and dividing by the per-channel standard deviation,

$$\text{standardised input}_{i,c,x,y} = \frac{x_{i,c,x,y} - \mu_c}{\sigma_c}. \tag{9}$$

Importantly "standardised input" has mean 0 and standard-deviation 1 for all channels. This is a bit restrictive: the neural network might perform better if the outputs have a different mean and standard deviation. Therefore, batchnorm finally multiplies by per-channel scale-factor, γ_c and adds a per-channel offset, β_c,

$$\text{output}_{i,c,x,y} = \gamma_c \text{ standardised input}_{i,c,x,y} + \beta_c. \tag{10}$$

Thus, the outputs of batchnorm have standard deviation γ_c and mean β_c.

There are lots of hypotheses for how batchnorm works, but I haven't really been convinced by any of them. Suffice it to say that batchnorm seems to stabilise training, and it seems to be *really* important for getting neural networks to work well. In fact, it gives perhaps the biggest boost to neural network performance of any modification (except perhaps data augmentation, which also makes a huge difference).

6 Classic convolutional neural networks

We're now in a position to understand the full architecture of some of the most classic convolutional neural networks (CNNs). Specifically, we're going to look at arguably the two historically most important two neural networks: LeNet and AlexNet.

LeNet was developed in Yann LeCun et al. in 1998, and was designed to recognise images of handwritten digits (MNIST). This dataset included 60,000 greyscale, 28×28 images of handwritten digits (we have already seen this dataset). The resulting system was deployed in the real-world to read millions of checks. LeNet was the first demonstration that neural networks could do interesting tasks.

AlexNet was developed by Alex Krizhevsky et al. in 2012. If we have to choose a network that started the "deep learning revolution", most people would choose AlexNet. AlexNet was designed to solve a much, much harder task: recognising one of 1000 objects in large (224 x 224 color images). Specifically, this dataset is known as ImageNet-1k (as it has 1000 object classes). This dataset has about one million training examples. AlexNet was about 50% better than the previous best method, based on hand-engineered features.

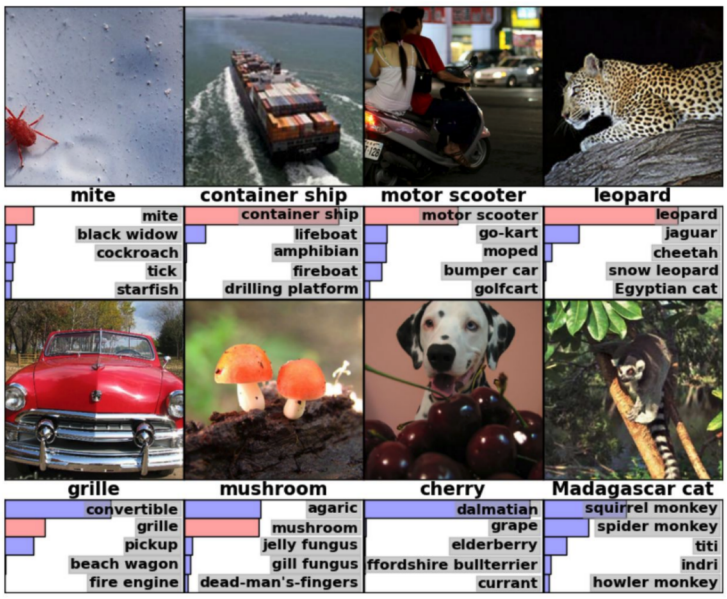


Figure 9: Examples of images from imagenet, along with their class labels (as given in the dataset), and example classifications from a neural network. Note that class-labels are often ambiguous (dalmation vs cherry, convertible vs grille vs pickup, mushroom vs agaric), so 100% accuracy likely isn't possible.

Lets zoom in a bit on the architecture (the details of these architectures is non-examinable; I'm not going to ask about the architecture of LeNet / AlexNet; but they are useful for illustrating what these networks typically look like).

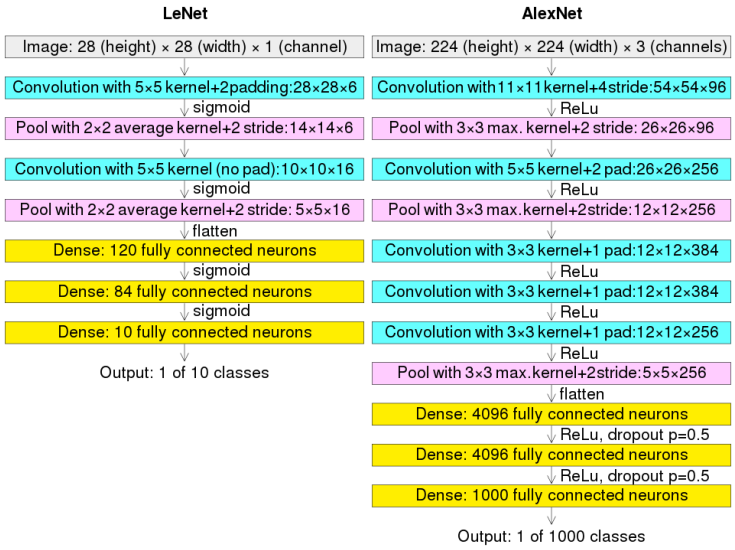


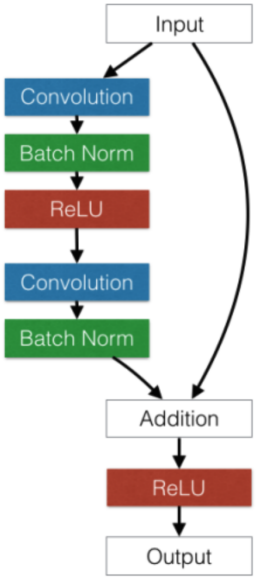
Figure 10: Each box is a fully-connected (dense), convolutional, or pooling layer. The size of the output is given in each box after the colon.

We can see the differences are surprisingly minimal. AlexNet's success is typically attributed to:

- Use of relu rather than sigmoid nonlinearities. As discussed previously, relu gives better backpropagation of gradients.
- AlexNet was one of the first times that GPUs were used to train neural networks, so they were able to use much more compute for training.
- Dropout for regularisation.
- An early variant of batchnorm after some of the nonlinearities (not shown on the diagram). This variant is no longer used, so is not discussed further in this course.

7 Modern convolutional neural networks: ResNets

Modern convolutional networks have a very similar structure. The only difference is the addition "skip connections". They're called "skip" connections, because skip over a number of layers in the network. Modern convolutional neural networks with skip connections are typically known as "ResNets" or "Residual Networks". A skip connection looks something like:



Note the arrow on the right that bypasses the two convolutional layers. In equations,

$$a_1 = \text{batchorm}(\text{conv}(\text{input}, ...)) \tag{11a}$$

$$a_2 = \text{batchorm}(\text{conv}(\text{relu}(a_1), ...)) \tag{11b}$$

$$\text{output} = \text{relu}(\underbrace{\text{input}+}_{\text{skip conn.}} a_2) \tag{11c}$$

Here, the ... denotes the parameters of the convolutional layer, i.e. the weights and biases.

There is a skip connection in the final expression, Eq. (11c), arising from the use of "input" in that expression. Using "input" directly means skipping over all the NN layers used in computing a_1 and a_2.

Don't worry about the precise details of exactly what order the operations go in: there are *loads* of different variants. The key idea is just that the skip connections bypass some network layers.

Most of the modern "backbone" networks for image tasks use some variant of a ResNet (i.e. a convolutional network with these skip connections.

ResNets seem to work really well, but it isn't really known why. One idea is that the skip connections help "signal" propagate through the network early on, and thereby help with training. But again, I'm not hugely convinced by this idea.