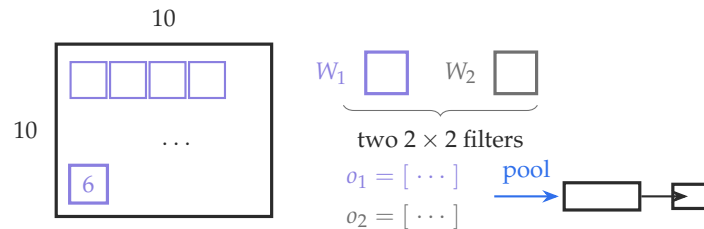


ConvNets (2)

DS 4440 · LECTURE 10

1. Where we left off

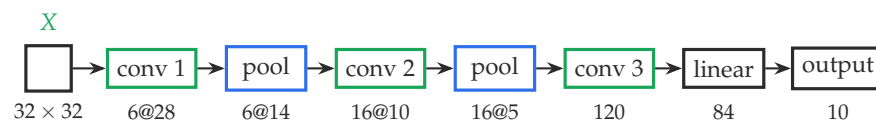
Two 2×2 filters swept over a 10×10 input, each giving its own feature map; pool each map and concatenate the results.



Design decisions: filter sizes · number of filters · padding · stride.

2. In practice: deep models dominate

Some history: **LeNet** (LeCun, 1998) had the architecture essentially right. Hardware and data had to catch up.

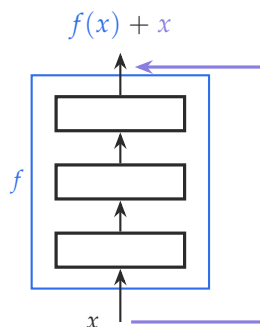


Every number follows from 5×5 kernels with no padding and 2×2 pooling: $32 - 5 + 1 = 28$, then $28/2 = 14$, then $14 - 5 + 1 = 10$, then $10/2 = 5$, and finally $5 - 5 + 1 = 1$ — which is why conv 3 emits a vector rather than a map.

In 2012 the time came: **AlexNet** won the ImageNet challenge. It was deeper and used **ReLU**, and it also leaned on GPUs and on dropout, which is §4 below. (See the Colab notebook.)

3. Residual connections

Training gets tricky with deeper nets. Adding paths that **skip** layers can help.



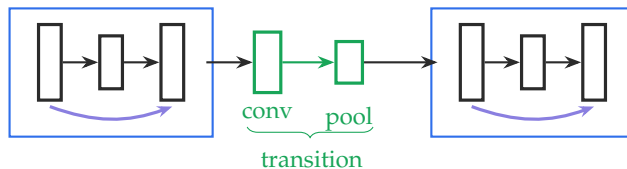
Can **add** (+) or **concatenate** ($\oplus$).

Adding needs the shapes to match; when they do not, the skip path carries a 1×1 convolution to fix them.

If we $\oplus$, dimensions may get out of hand.

Why a skip helps: it gives the gradient a route past f with no weight matrices and no σ' factors on it (Lecture 5, §8). Even when the deep path is saturated, the identity route keeps error flowing to the layers below.

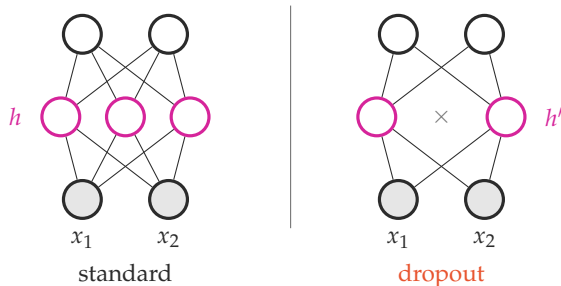
DenseNet and similar use **transition layers**, which downsample the feature maps (e.g. via pooling) between blocks.



4. Dropout

Another standard trick while training deep nets: inject **noise** during training as a regularization technique.

$$h' = \begin{cases} 0 & \text{with probability } p, \\ \frac{h}{1-p} & \text{otherwise.} \end{cases}$$



Why the $1/(1-p)$? So the expected activation is unchanged:

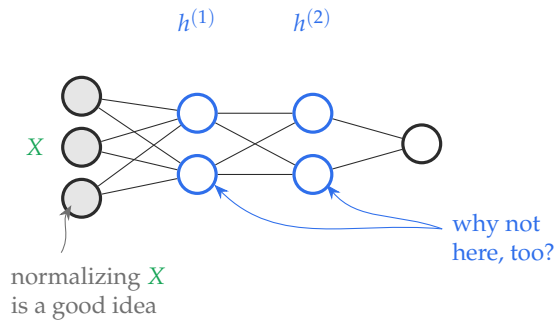
$$\mathbb{E}[h'] = p \cdot 0 + (1-p) \cdot \frac{h}{1-p} = h.$$

That matters because at **test time we do not drop anything** — we use the whole network. Scaling during training is what keeps the two regimes consistent.

Why this regularizes: a unit cannot count on any particular other unit being there, so the network cannot build fragile chains of co-adapted features. It is also, loosely, like training an ensemble of smaller networks that share weights.

5. Batch norm

When we covered optimizers we discussed **normalizing inputs** for gradient descent. Why not here, too?



With **batch norm** we do this per batch. Following Lecture 6, write $\tilde{h}^\ell$ for the pre-activation at layer ℓ , so $h^\ell = a(\tilde{h}^\ell)$. Over the m examples in the batch,

$$\mu^\ell = \frac{1}{m} \sum_{j=1}^m \tilde{h}_j^\ell, \quad (\sigma^\ell)^2 = \frac{1}{m} \sum_{j=1}^m (\tilde{h}_j^\ell - \mu^\ell)^2, \quad \hat{h}^\ell = \frac{\tilde{h}^\ell - \mu^\ell}{\sqrt{(\sigma^\ell)^2 + \epsilon}},$$

with ϵ there for numerical reasons. That gives zero mean and unit variance. Batch norm then adds flexibility by letting the result be rescaled and shifted:

$$\tilde{h}_{\text{norm}}^\ell = \underbrace{\hat{\gamma}^\ell}_{\text{learned}} \odot \hat{h}^\ell + \underbrace{\hat{\beta}^\ell}_{\text{learned}}$$

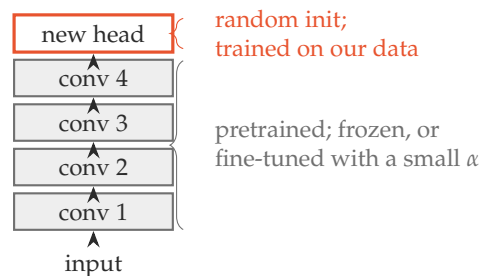
$\hat{\gamma}$ and $\hat{\beta}$ are parameters like any others, one pair per feature, and they are trained by backprop along with the weights.

One practical note: at **test time there is no batch** to compute μ and σ from, so we use running averages collected during training. This is why a model can behave differently in `train()` and `eval()` mode.

One consequence worth knowing: μ and σ are estimated from the batch, so with a small batch they are noisy, and each example's normalization depends on whichever others happen to sit alongside it. **Layer norm** avoids both by normalizing across the *features* of a single example rather than across the batch, which is what sequence models tend to use.

6. Transfer learning

Training a deep net from scratch needs a lot of labelled data. Most of the time we do not have it — and most of the time we do not need it, because somebody has already trained a network on ImageNet, and the early layers of that network are not especially about ImageNet.



The recipe:

1. Take a network pretrained on a large dataset.

2. Throw away its final layer — it predicts the wrong classes — and put a fresh one in its place, sized for our labels.
3. **Freeze** the earlier layers, or fine-tune them with a much smaller learning rate than the new head.

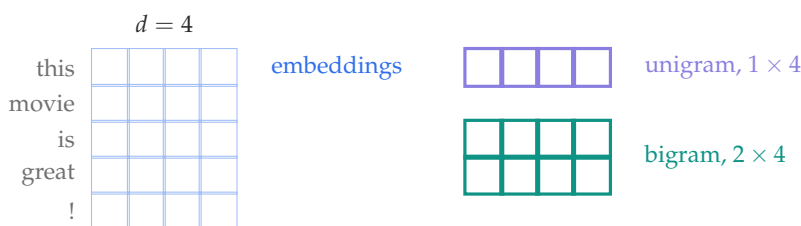
Why this works is the representation argument from Lecture 3, one layer at a time. Early filters learn edges and blobs, which are not specific to the task that trained them; later layers compose those into something closer to the labels. So the early layers transfer and the last ones do not.

Freeze or fine-tune is the same decision we faced with embeddings in Lecture 8, for the same reason. Freezing fixes the pretrained features and fits only the head, which is what we want when the labelled set is small — there is little to overfit with. Fine-tuning adapts the features to our data, which helps when we have enough of it, and wants a small α so the first noisy gradients from a randomly initialized head do not wreck weights that were already good.

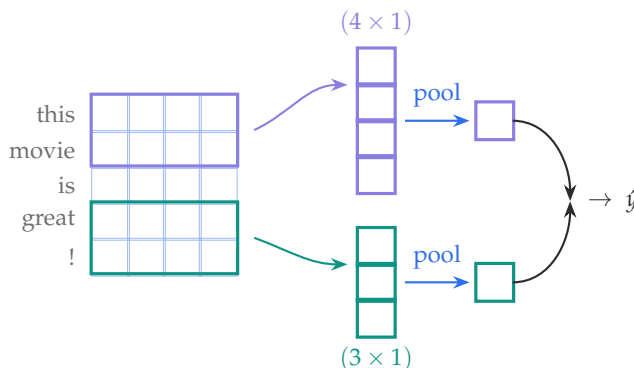
Starting from a pretrained network is the default for images. Training from scratch is for when the data looks nothing like anything already pretrained on, or when there is a great deal of it.

7. CNNs for text

ConvNets are mostly used for images, but the same machinery applies to a sentence once we replace each word with its embedding (the 9/30 lecture). The input is an $n \times d$ matrix: one row per token, d columns of embedding.



A filter spans the *full* embedding width, so its only free dimension is how many tokens it covers — which makes the filter size a choice of n -gram. Sliding it down the sentence gives one feature map per filter, and pooling each map gives a fixed-size vector however long the sentence is.



With five tokens, a bigram filter (2×4) gives $5 - 2 + 1 = 4$ positions and a trigram filter gives 3. (See the Colab notebook.)

Bonus: adversarial examples

Back in Lecture 4 we computed $\partial\mathcal{L}/\partial x$ — the gradient with respect to the *input* — and then set it aside, because x is data and we do not optimize it. Suppose we use it anyway.

$$\begin{array}{ccccc} \boxed{} & + \epsilon & \boxed{} & = & \boxed{} \\ x & & \text{sign}(\partial\mathcal{L}/\partial x) & & x' \\ \text{"panda" 0.94} & & & & \text{"gibbon" 0.99} \end{array} \quad \begin{array}{l} \text{indistinguishable} \\ \text{to a person} \end{array}$$

Gradient descent moves the *weights* to lower the loss. Here we hold the weights fixed and move the *input* to *raise* it:

$$x' = x + \epsilon \cdot \text{sign}\left(\frac{\partial\mathcal{L}}{\partial x}\right),$$

with ϵ small enough that no pixel changes by much. Taking the sign means every pixel moves by the same small amount, in whichever direction hurts — so the perturbation stays tiny per pixel while adding up across all of them. One step is often enough to flip the prediction, usually with higher confidence than the original.

Nothing here is exotic: it is the same backward pass, stopped one node earlier. The machinery that trains the network also tells you how to break it.

The model was 94% confident, and a change you cannot see makes it 99% confident of something else. High confidence is not evidence that a network is looking at what we would look at.

This is why adversarial robustness is its own research area, and it is a reason to be careful about deploying these models where someone has an incentive to fool them.