CSE 592: Foundations of AI — University of Michigan

Lecture 11: Deep Learning



Gabriel Poesia
Assistant Professor, CSE
University of Michigan, Ann Arbor
poesia@umich.edu
https://gpoesia.com

Plan

1. Learning rate schedules
2. Making depth work
3. Learning from images
4. Generating images
5. Generalization

Where we left off

🧠 **Recall**

Backpropagation computes gradients.
Optimizer uses them to update weights.

AdamW: momentum + adaptive updates + weight decay.

Now we can train! But what models (architectures)?

Large, deep models bring new training challenges.

? Question

Does adding layers give us a better model?

Today: techniques for training deep models, then learning from images.

Learning rate schedules

Learning rate schedule

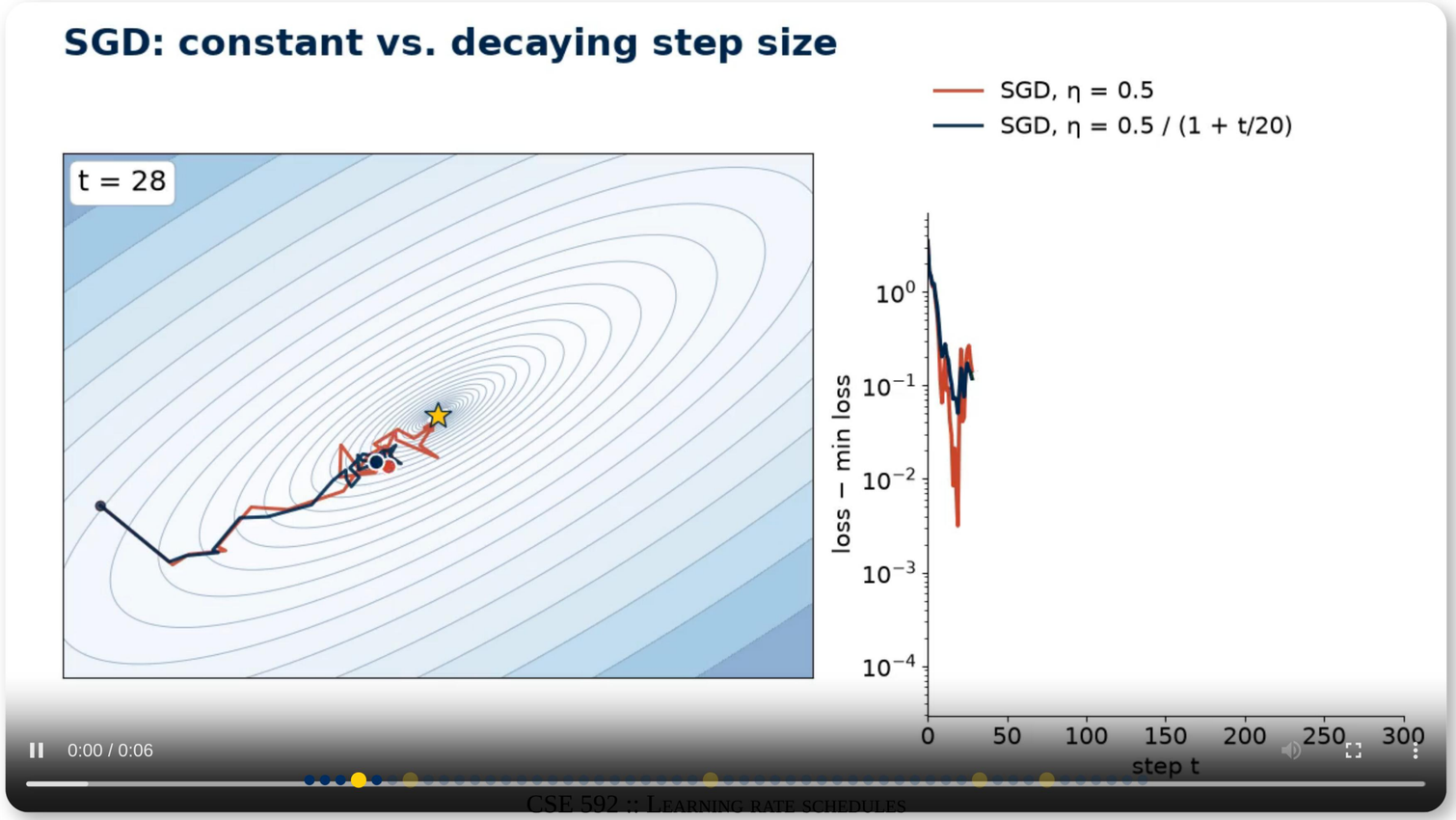
Recall: minibatch SGD/AdamW uses noisy gradient estimates.

Constant η: noisy gradients keep moving the weights.

Near a minimum, smaller steps can help.

Idea: decay the learning rate during training.

Let's see an SGD example:



Warmup, then decay

A **learning rate schedule**: choose η_t at iteration t.

Common pattern: start small, increase briefly, then decay slowly.

Warmup: reach $\eta_{\max}$ over first T_w iterations.

$$\eta_t = \eta_{\max} \frac{t}{T_w}, \qquad 0 \le t \le T_w$$

Then use **cosine decay**, until the final iteration T.

Progress: $p = \frac{t - T_w}{T - T_w}$.

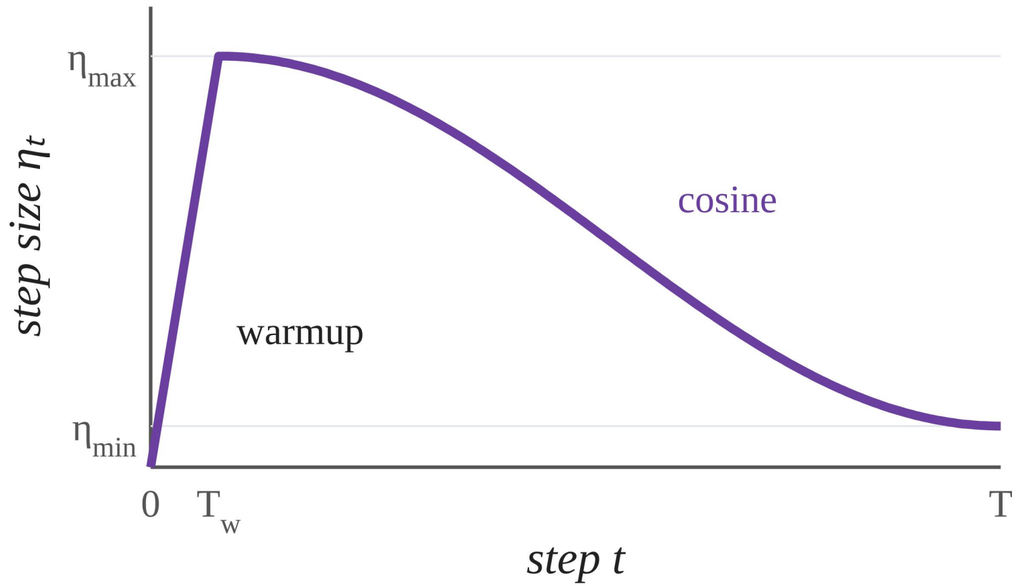
$$\eta_t = \eta_{\min} + (\eta_{\max} - \eta_{\min}) \frac{1 + \cos(\pi p)}{2}$$

Q: What rates do we get at $p = 0$ and $p = 1$?

A: $\eta_{\max}$ and $\eta_{\min}$.

Warmup can stabilize early updates.

Decay: smaller steps over time.



Making depth work

Going deeper

A **multi-layer perceptron (MLP)** repeats two operations:

- Linear transformation + bias: $z_\ell = W_\ell h_{\ell-1} + b_\ell$.
- Activation function: $h_\ell = \sigma(z_\ell)$, e.g., ReLU.

The linear stage connects every input to every unit: **fully connected**.

Start with $h_0 = x$. Repeat for L layers.

Each output is the next layer's input.

Final layer makes the prediction.

Is deeper always better?

Suppose we have a 20-layer ReLU MLP.

Insert 36 hidden layers of the same width.

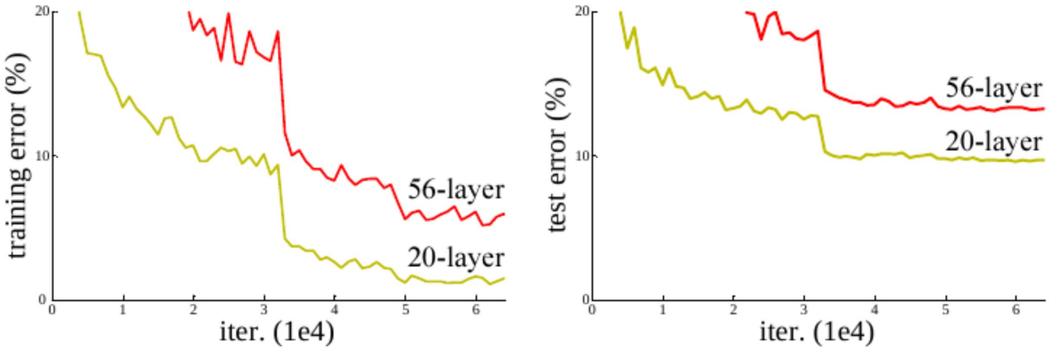
Q: Can the 56-layer MLP represent every function of the 20-layer MLP?

A: Yes! Set $W = I, b = 0$ in each extra layer.

ReLU hidden inputs are nonnegative: $\text{ReLU}(Ih) = h$.

But training the deeper model can give **higher error**:

Q: Is this approximation error or estimation error?



He et al., Deep Residual Learning for Image Recognition (2016) — Fig. 1. Image-classifier experiment (layers explained later). Training (left), test (right). CIFAR-10: small images, 10 classes.

Why did the error increase?

> 🧠 **Recall**
>
> **Approximation error:** how good is the best predictor in the class?
>
> **Estimation error:** how well do we choose a predictor in that class?

Q: Which explains the higher test error?

Larger hypothesis class $\Rightarrow$ approximation error cannot increase.

So estimation error increased by more than approximation error decreased.

> ⚠ **Careful**
>
> **Training error increased too!**
>
> The optimizer didn't even match the smaller model's training performance.

Good weights exist, but finding them is hard.

What makes optimization harder with depth?

Vanishing and exploding gradients

What makes optimization harder with depth?

Backpropagation multiplies by a local derivative at each layer.

Let's think of a simple case: just one scalar gradient.

Each layer multiplies it by the same number a.

$$\text{input gradient} = a^L \cdot \text{output gradient}$$

a	a^{50}	**Effect**
0.9	0.005	**Vanishing:** tiny gradients in early layers
1.1	117	**Exploding:** very large gradients

In an MLP: matrix products, not just scalars.

Different directions can shrink or grow.

The forward pass has a similar problem.

How can initialization keep activations at a useful scale?

Initialization

Identical hidden units learn the same thing. So we initialize with random weights.

Let's try to control the variance of our activations at initialization?

> 🧠 **Recall**
> $\mathrm{Var}(X) = \mathbb{E}[X^2] - \mathbb{E}[X]^2$.
> If $\mathbb{E}[X] = 0$, then $\mathrm{Var}(X) = \mathbb{E}[X^2]$.

One unit: $z_j = \sum_{i=1}^{d} W_{ji} h_i$. No bias for now.

- Assuming independent weights: mean 0, variance s^2, independent of h.
- Inputs have $\mathbb{E}[h_i^2] = q$. Q: What is $\mathbb{E}[z_j^2]$?

$$\mathbb{E}[z_j^2] = \sum_{i=1}^{d} \mathbb{E}[W_{ji}^2]\mathbb{E}[h_i^2] = d\, s^2 q$$

Output variance grows by ds^2!

To keep it constant q, we can set $s^2 = 1/d$.

More inputs $\Rightarrow$ smaller individual weights.

Glorot & Bengio, Understanding the Difficulty of Training Deep Feedforward Neural Networks (2010). Forward-scale argument.

What about ReLU?

Our MLP passes $\mathrm{ReLU}(z_j)$ to the next layer.

Suppose z_j is distributed symmetrically around 0.

Q: What happens to its average squared size after ReLU?

$$\mathbb{E}[\mathrm{ReLU}(z_j)^2] = \tfrac{1}{2}\mathbb{E}[z_j^2] = \tfrac{1}{2} d\, s^2 q$$

ReLU removes the negative half of the squared size.

> 💡 **Insight**
> To preserve variance q after ReLU: initialize with $s^2 = 2/d$.

He initialization: $W_{ji} \sim \mathcal{N}(0, 2/d)$.

d: number of inputs to the unit.

This controls scale at initialization.

He et al., Delving Deep into Rectifiers (2015).

Small differences add up

Let's test the initialization we just derived.

Same ReLU MLP: 50 layers, 512 units each.

Q: What if we use $1/d$, $2/d$, or $3/d$ for the weight variance?

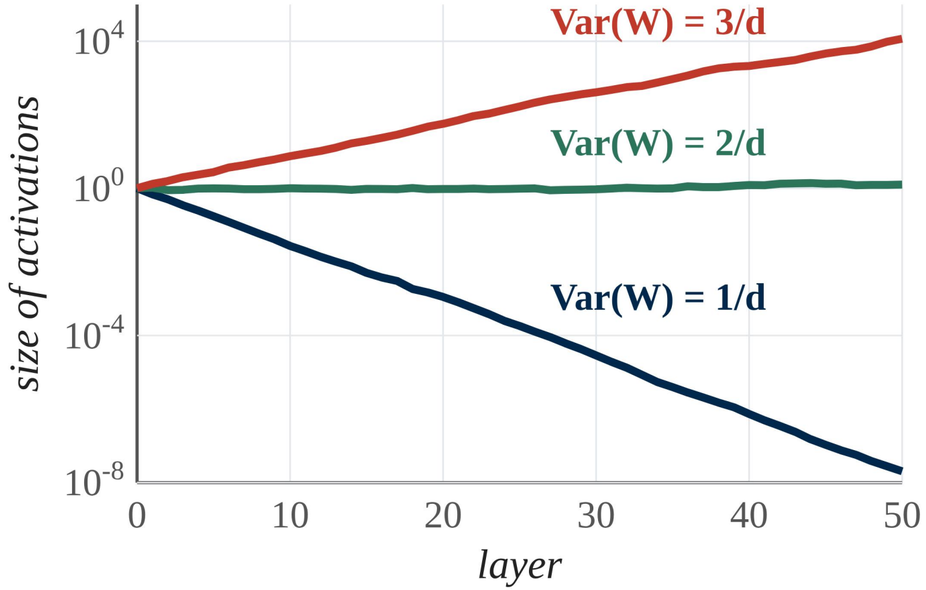
$1/d$: average squared size halves at each layer.

$2/d$: roughly preserves the scale.

$3/d$: squared size grows by 1.5 per layer.

Same architecture. Different initialization.

Very different scales, before any training!



CPU simulation at random initialization. Vertical axis: root mean square activation size, on a log scale.

Keeping the mean and variance stable

Initialization controls activation scale at the start.

But training changes the weights! How can we avoid vanishing / explosion during training too?

Problem: activation means and variances can drift.

> **? Question**
> Can we keep mean and variance stable as we add layers?

High-level idea: compute activations, then subtract the mean & divide by standard deviation.

$$\hat{z} = \frac{z-\mu}{\sqrt{v+\epsilon}}$$

μ: mean. v: variance.

Small $\epsilon > 0$: avoid division by zero.

Nonconstant values: mean 0, variance ≈ 1.

But which values do we average over?

Batch normalization

One idea: take all of one layer's activations in $Z \in \mathbb{R}^{B \times d}$.

B examples (rows), d features (columns).

BatchNorm: normalize each feature across the batch.

$$\mu_j = \frac{1}{B}\sum_{i=1}^{B} Z_{ij}, \qquad v_j = \frac{1}{B}\sum_{i=1}^{B}(Z_{ij} - \mu_j)^2$$

Separate mean and variance for each feature.

$$\hat{Z}_{ij} = \frac{Z_{ij} - \mu_j}{\sqrt{v_j + \epsilon}}$$

Then allow a learned scale γ_j and offset β_j:

$$Y_{ij} = \gamma_j \hat{Z}_{ij} + \beta_j$$

Train γ_j, β_j along with the network weights.

Ioffe & Szegedy, Batch Normalization (2015).

BatchNorm: an example

One layer: 4 examples, 3 features each.

Q: Mean and variance of the highlighted column?

$$\mu_1 = 4, \qquad v_1 = \frac{9+1+1+9}{4} = 5$$

For simplicity using $\gamma_1 = 1$, $\beta_1 = 0$, ignoring ϵ.

Normalized values:

$$\frac{[-3,-1,1,3]}{\sqrt{5}} \approx [-1.34, -0.45, 0.45, 1.34]$$

Same for each column.

Each normalized column has mean 0, variance 1.

	F1	F2	F3
Ex. 1	1	3	5
Ex. 2	3	5	7
Ex. 3	5	7	9
Ex. 4	7	9	11

Suppose the same example came with a different batch.

Q: Would its normalized values change?

A: Yes!

BatchNorm at prediction time

Training: compute statistics from the current minibatch.

Q: What if we predict for just one example?

Consider the MLP again: we'd have one value per feature.

Batch variance $= 0$. Subtracting the mean gives 0 too. Oops!

Solution: track running means and variances during training.

Prediction: use those stored statistics. Now inference is deterministic.

> ⚠ **Careful**
>
> BatchNorm needs distinct **training** and **evaluation** modes.

> ? **Question**
>
> Can we normalize an example without using other examples?

Layer normalization

Alternative: normalize one example using *its own features*.

LayerNorm for $z \in \mathbb{R}^d$:

$$\mu = \frac{1}{d}\sum_{j=1}^{d} z_j, \qquad v = \frac{1}{d}\sum_{j=1}^{d}(z_j - \mu)^2$$

$$\hat{z}_j = \frac{z_j - \mu}{\sqrt{v+\epsilon}}, \qquad \mathrm{LN}(z)_j = \gamma_j \hat{z}_j + \beta_j$$

Same normalization formula, but different averaging axis.

Each example gives its own statistics.

Same rule in training and test timeh.

We'll use LayerNorm for language models next lecture.

Ba, Kiros & Hinton, Layer Normalization (2016).

LayerNorm: an example

Same activations. Now normalize *one row*.

Example 1: $z = [1, 3, 5]$.

Again: $\gamma_j = 1$, $\beta_j = 0$. Ignore ϵ.

Q: Mean and variance across these features?

$$\mu = 3, \qquad v = \frac{4+0+4}{3} = \frac{8}{3}$$

$$\text{LN}(z) = \frac{[-2, 0, 2]}{\sqrt{8/3}} \approx [-1.22, 0, 1.22]$$

Now examples in the same batch are independent, again.

	F1	F2	F3
Ex. 1	1	3	5
Ex. 2	3	5	7
Ex. 3	5	7	9
Ex. 4	7	9	11

BatchNorm and LayerNorm

Same activation matrix, different normalization axes:

	BatchNorm	LayerNorm
Values averaged	One feature (column)	One example (row)
Each example depends on	Other examples in training batch	Own feature values
At prediction time	Running statistics from training	Own statistics

Similar idea: normalize activations + learn a scale and offset.

Many other normalization layers in the literature, but these are still very popular.

Residual connections

In a deep neural net, each layer/block is computing a new representation given the previous one.

In principle, more layers give more expressivity (if useless, some layers could become the identity).

But optimizers seem to struggle to find those weights!

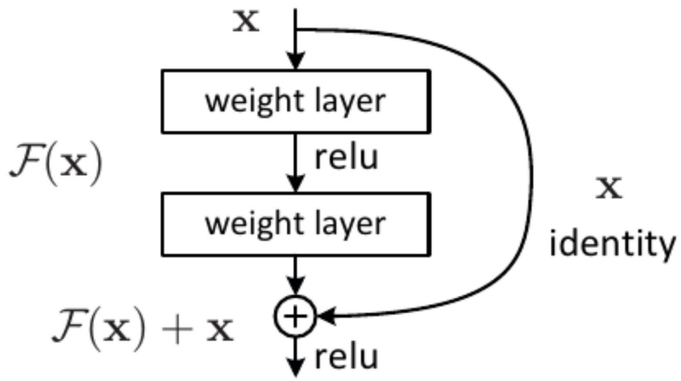
One idea: learn a **change** to the input, rather than full transformation.

$$h_{\ell+1} = h_\ell + F_\ell(h_\ell)$$

Q: What should F_ℓ output to compute the identity?

A: 0. The shortcut carries h_ℓ forward.

$$\frac{\partial h_{\ell+1}}{\partial h_\ell} = I + \frac{\partial F_\ell(h_\ell)}{\partial h_\ell}$$



He et al., Deep Residual Learning for Image Recognition (2016) — Fig. 2. The original block also applies ReLU after the addition.

Will be used in several architectures we'll see.

> 💡 **Insight**
> The identity term gives gradients a direct path from the loss, even from far away!

Dropout

We can train deeper models. But they can still overfit.

One idea: don't let units "over-rely" on the same input features.

Dropout: randomly set some activations to 0 in training.

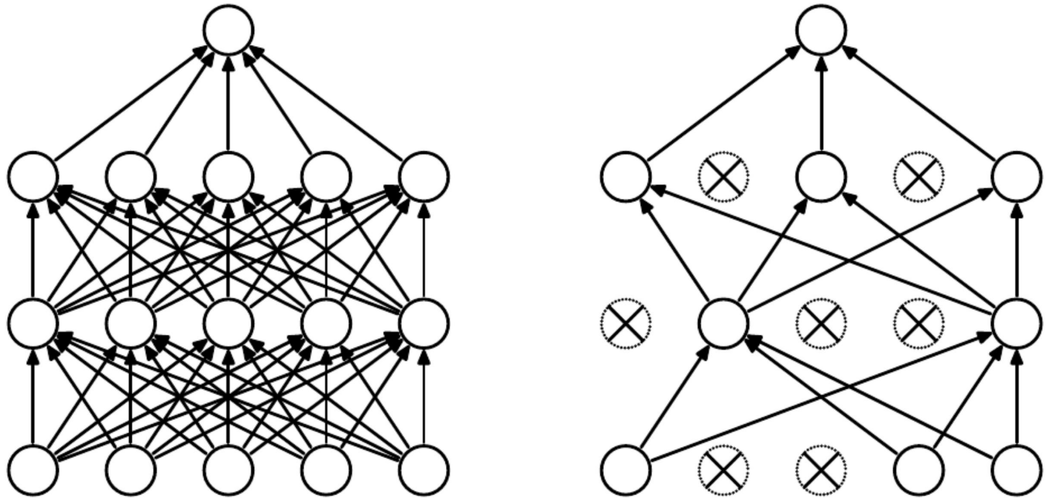
Choose a new random set each pass.

Remaining units must still make useful predictions.

Like weight decay: regularization to help generalization.

Q: What should we do at prediction time?

First, let's work out what happens to the activations.



(a) Standard Neural Net (b) After applying dropout.

Srivastava et al., Dropout: A Simple Way to Prevent Neural Networks from Overfitting (2014) — Fig. 1. Crossed-out units don't contribute on this training pass.

Dropout: training and prediction

Activations: $h = [2, 4, 6, 8]$.

Suppose we drop each with probability $p = 0.5$.

Suppose the mask is $m = [1, 0, 1, 0]$.

Q: What remains after dropping those activations?

A: $[2, 0, 6, 0]$.

Each activation is kept only half the time.

So its expected value is half as large.

Need to compensate by doubling the remaining activations.

$$\tilde{h} = [4, 0, 12, 0]$$

In general: drop probability $0 \leq p < 1$.

$m_j = 1$ with probability $1 - p$; otherwise 0.

$$\tilde{h}_j = \frac{m_j h_j}{1-p}, \qquad \mathbb{E}[\tilde{h}_j \mid h_j] = h_j$$

Prediction: use all activations h directly.

Srivastava et al. (2014). The paper also describes the equivalent convention of rescaling at prediction time.

What do we train these models on?

Big bag of techniques to train deep models! Let's see some actual models now.

What do we train them on?

Input modality: kind of data.

Images, text, audio, video, ...

For each modality, need to choose:

- A representation of the input (as input vectors).
- An architecture that makes predictions.

Output: a prediction, or features for another model.

Multimodal: multiple input modalities.

Example: image + question $\rightarrow$ answer.

We'll focus on images today.

Learning from images

How do we represent an image?

A color image is a grid of pixels.

Each pixel: red, green, blue (RGB).

224×224 pixels $\Rightarrow 224 \cdot 224 \cdot 3 = 150{,}528$ numbers.

First idea: just flatten into a vector & train an MLP?

First layer with 1000 units: **150 million weights**!

Suppose we shift an object one pixel right.

Q: Does the network reuse the same detector weights?

No! New positions connect to different weights.

Sharing across positions isn't built into the MLP.

We'd like to use two facts about images:

- Nearby pixels form patterns.
- The same pattern can occur anywhere.

Convolutions

One operation that captures these properties: convolution

Let's do 1D first. Start with one row of pixels: $x_1, \ldots, x_n$.

At position t: combine three neighbors.

$$z_t = a\,x_{t-1} + b\,x_t + c\,x_{t+1}$$

Move the window. Reuse a, b, c.

This is a **convolution**.

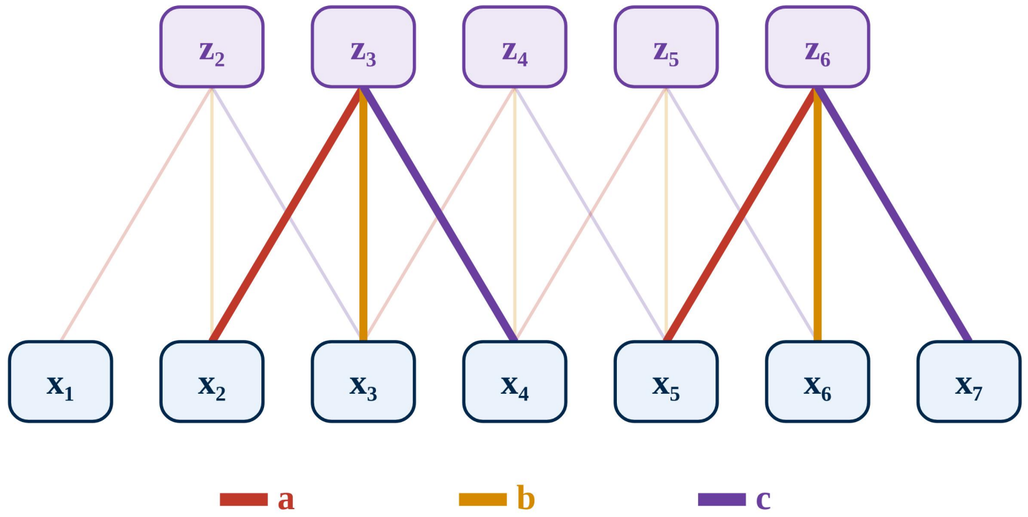
Only three learned weights, for any input length.

Q: Shift the input. What happens to the output?

A: It shifts too (away from boundaries).

Translation equivariance.

For images: use a 2D window.



A convolution by hand

Suppose a 3×3 grayscale image, 2×2 learned **filter**, no bias for now.

Stride 1: move one pixel at a time.

1	2	0
0	3	1
2	0	1

Input x

1	0
0	-1

Filter K

$\longrightarrow$

-2	1
0	2

Output z

Q: What is the output for the highlighted window?

$$1 \cdot 1 + 2 \cdot 0 + 0 \cdot 0 + 3 \cdot (-1) = -2$$

Move the filter for the other outputs.

Then ReLU: $[[-2, 1], [0, 2]] \mapsto [[0, 1], [0, 2]]$.

Channels and filters

RGB: three grids of values, called **channels**.

One 3×3 filter spans all three channels.

$3 \cdot 3 \cdot 3 = 27$ weights.

Sum over the window and input channels.

Outputs across positions form a **feature map**.

This is one channel for the next layer.

Want 64 output channels? Learn 64 filters (also called kernels).

$$\text{weights} = k^2 C_{\text{in}} C_{\text{out}} = 3^2 \cdot 3 \cdot 64 = 1728$$

k: filter width. Optional bias per output channel.

Backprop learns what the filters do directly from the task's loss!

Seeing a larger part of the image

Can local filters recognize a whole object?

Deep neural nets: we stack convolutions across layers.

Let's track what one unit "sees" through 3×3 kernels (stride 1).

One layer: 3×3 input pixels.

The second layer combines neighboring first-layer outputs.

Q: How much of the original input can it see?

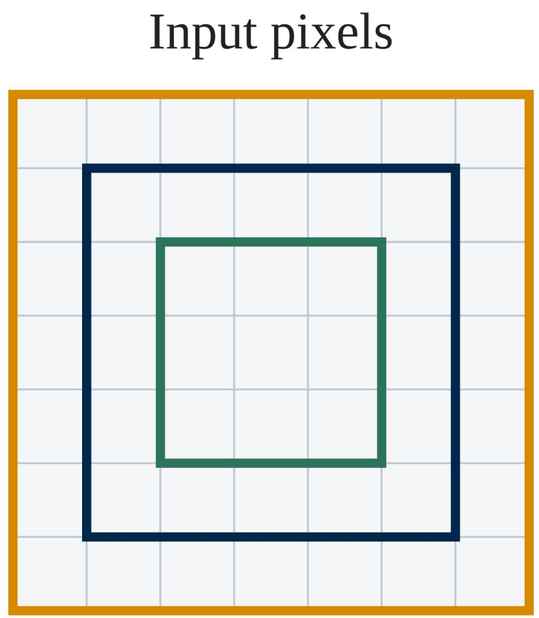
Two layers: 5×5 input pixels.

One extra border pixel on each side.

Q: What about three such layers?

Three layers: 7×7 input pixels.

Receptive field: input region that can affect a unit.

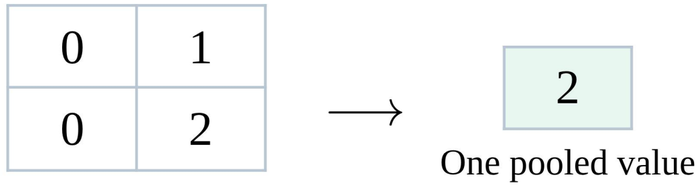


Reducing spatial resolution

We can also shrink the feature maps.

Max pooling: replace each window with its largest value.

Reuse our convolution output after ReLU:



One 2×2 window

One pooled value

Q: What is the pooled value?

A: $\max\{0, 1, 0, 2\} = 2$.

Nonoverlapping windows give fewer output positions.

Q: Can we recover the four input values from 2?

A: No. Pooling loses spatial detail.

Later layers often have fewer positions and more channels.

Fewer positions can reduce computation in later layers.

BatchNorm extends to image layers too:

One set of statistics per channel, over examples *and positions*.

Then learn a scale and offset per channel.

Even one image can have many values per channel.

From features to class probabilities

Last hidden layer: features $h(x)$. Predict one of C classes.

Linear layer: class scores $s = Wh(x) + b \in \mathbb{R}^C$.

Softmax: positive probabilities summing to 1.

$$p_\theta(y = c \mid x) = \frac{e^{s_c}}{\sum_{k=1}^{C} e^{s_k}}$$

Q: Scores $[2, 0, 1]$. Probability of class 1?

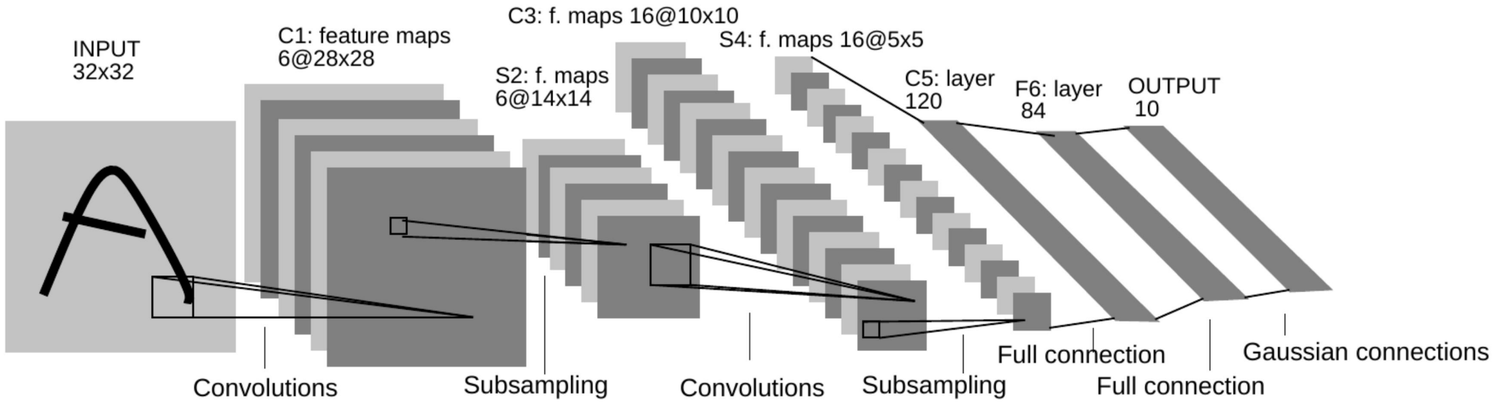
A: $e^2/(e^2 + 1 + e) \approx 0.665$.

> 🧠 **Recall**
>
> Recall logistic regression: minimize $-\log p_\theta(y \mid x)$.
>
> **Cross-entropy:** negative log-probability of the correct label.

Backprop trains the classifier *and* the feature layers.

For language models: classes are possible next words/tokens.

LeNet-5: a complete image classifier



LeCun et al., Gradient-Based Learning Applied to Document Recognition (1998) — Fig. 2. LeNet-5.

LeNet-5: a **convolutional neural network (CNN)** from 1998.

Convolution → activation → downsampling. Repeat, then make prediction (output layer).

Last layer is a linear classifier, but on **learned features**.

Features adapt to the task.

Some older choices: used smooth activations, trainable averaging.

Data augmentation

Convolution shares filters across positions.

Which other image changes should preserve the class?

One idea: change the image somehow without changing the label

A way to create additional data, make classifier robust to those changes

Data augmentation: create data by label-preserving transformations.

For object photos, for example:

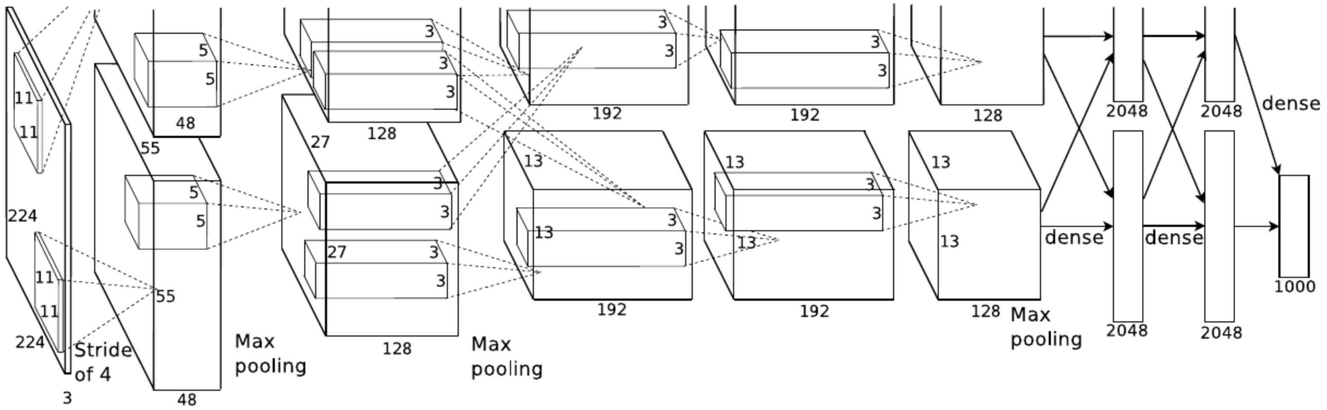
- Random crops.
- Horizontal flips.
- Small color changes.
- Small rotations.

Q: What if we rotate a handwritten 6?

> ⚠ **Careful**
>
> Transformations must preserve the label *for this task*.
>
> A rotated 6 might become a 9.

Can improve generalization / data efficiency if done properly.

AlexNet: scaling up CNNs



Krizhevsky, Sutskever & Hinton, ImageNet Classification with Deep Convolutional Neural Networks (2012) — Fig. 2. The two branches show the original two-GPU split.

AlexNet scales up CNNs to the **ImageNet 2012** task: 1000 object classes.

Training ingredients: GPUs, ReLUs, augmentation, dropout.

Top-5 error: true class missing from the five highest scores.

ImageNet 2012 top-5 test error:

AlexNet: 15.3%. Next best: 26.2%.

Result often credited with starting the deep learning craze (huge breakthrough at the time)

Trained just on class labels. What features did it learn?

First-layer filters

AlexNet's first layer.

Each tile: one learned $11 \times 11 \times 3$ filter.

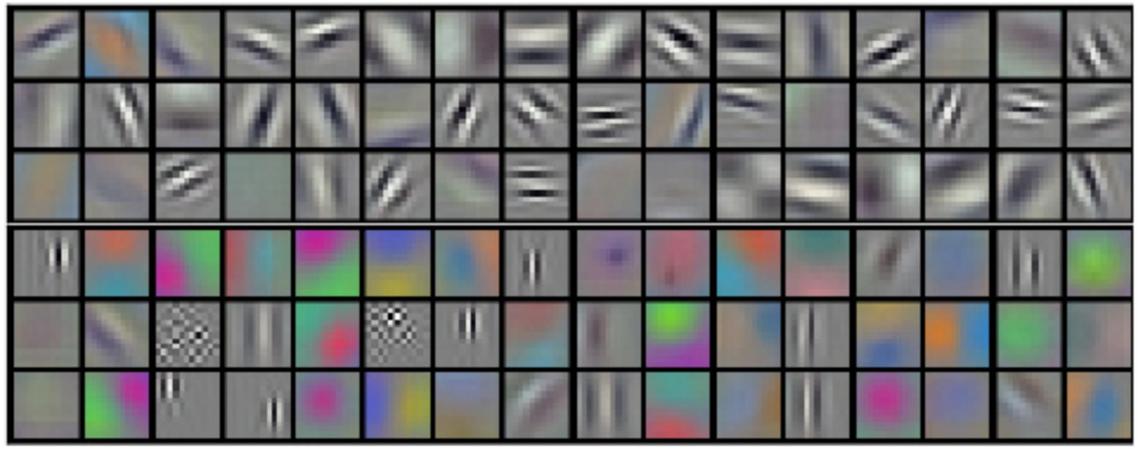
Q: What patterns do you recognize?

Edges at different angles. Color changes. Textures.

Training labels: image classes.

No labels for edges!

Useful filters help later layers classify images.



Krizhevsky et al., ImageNet Classification with Deep Convolutional Neural Networks (2012) — Fig. 3. These are weights, not activations for one image.

Intermediate layers

First-layer filters have three channels: RGB.

Later filters act on learned feature maps, so harder to visualize.

One idea: pick one channel, run validation images.

Find patches with large activations.

Project selected activations back to image space.

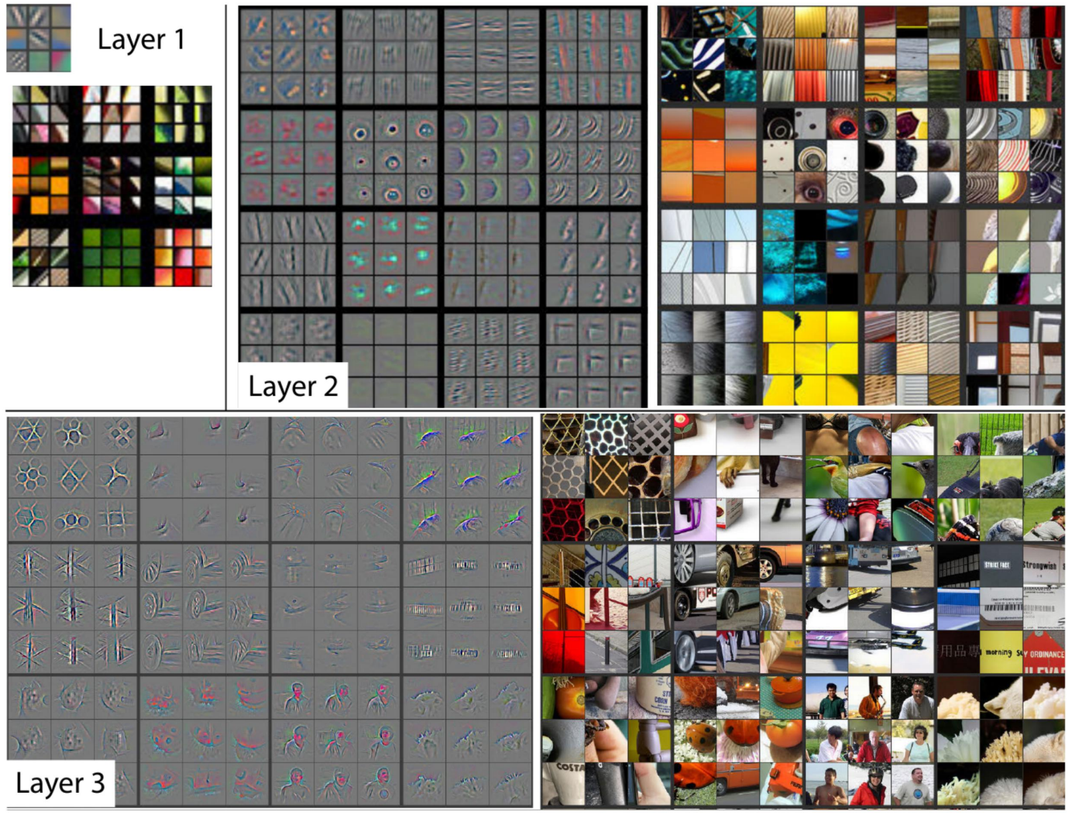
Q: What changes from layer 1 to layers 2 and 3?

Later layers: combinations of edges and textures.

Larger image regions.

Layers 2–3: tiles on gray backgrounds reconstruct activations.

Color tiles: original patches with strong activations.



Zeiler & Fergus, Visualizing and Understanding Convolutional Networks (2014) — Fig. 2, layers 1–3. Layer 1 shows filters; later layers show feature reconstructions and patches.

Deeper features

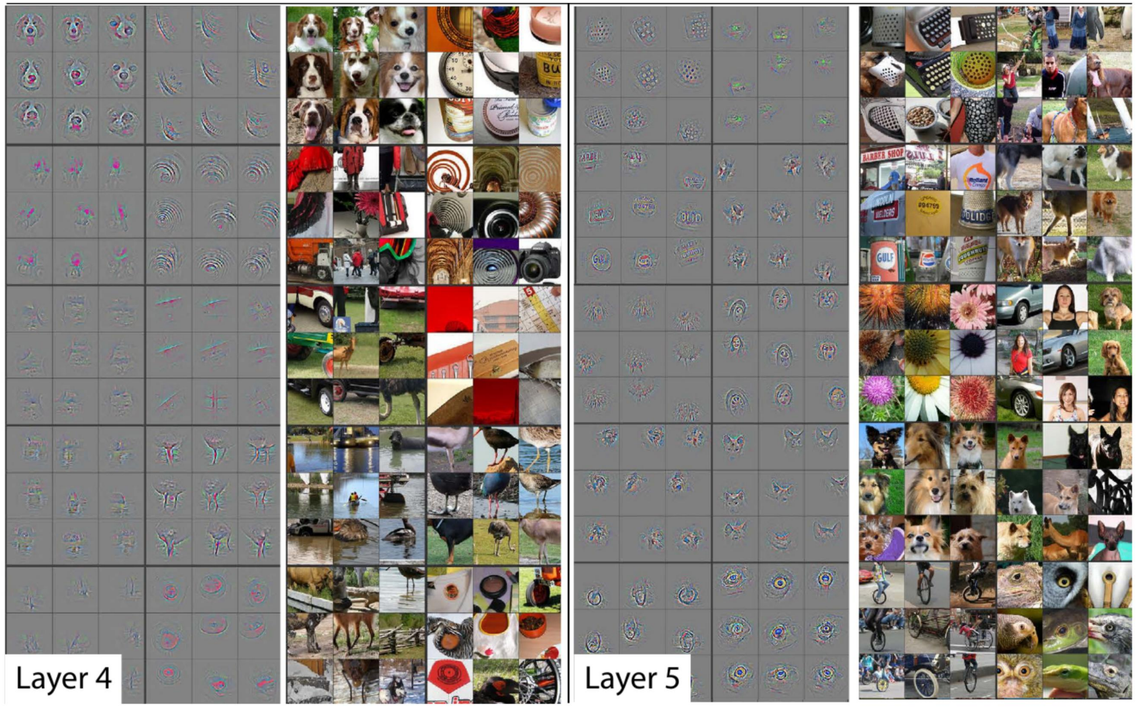
Same CNN. Now layers 4 and 5.

Q: Which image patches activate the same channel?

Shared patterns: faces, wheels, animal parts.

Different patches, similar learned features.

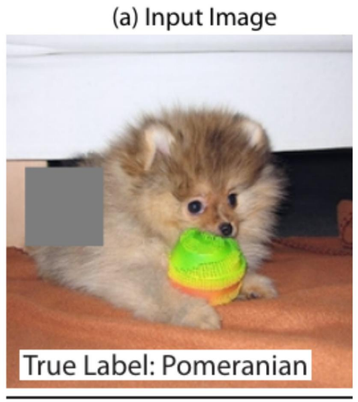
> ⚠ **Careful**
> One channel needn't mean one human concept.
> Features work together.



Zeiler & Fergus, Visualizing and Understanding Convolutional Networks (2014) — Fig. 2, layers 4–5. Each group pairs reconstructions with original patches.

Does the classifier actually use the object?

Let's test which pixels affect the final prediction.

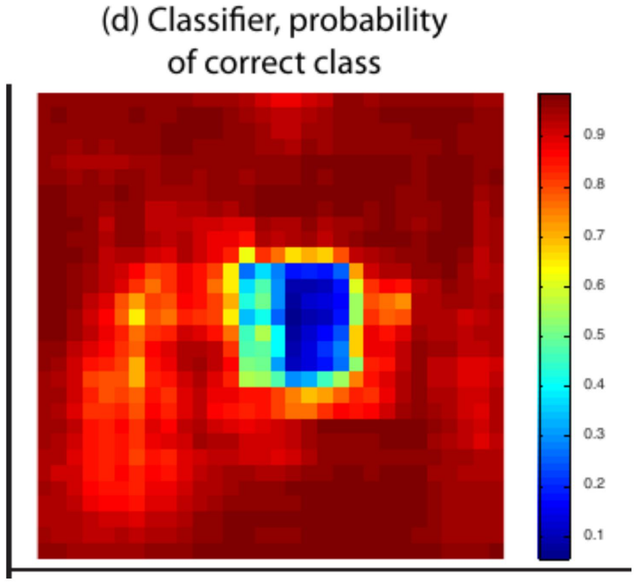


Zeiler & Fergus, Visualizing and Understanding Convolutional Networks (2014) — Fig. 7(a), first row. The gray square is moved across the image.

Cover a small square. Move it across the image.

Q: Where should the probability of “Pomeranian” drop most?

Now inspect the occlusion results:



Zeiler & Fergus, Visualizing and Understanding Convolutional Networks (2014) — Fig. 7(d), first row. Color at a position: class probability when that region is covered. Blue: low probability.

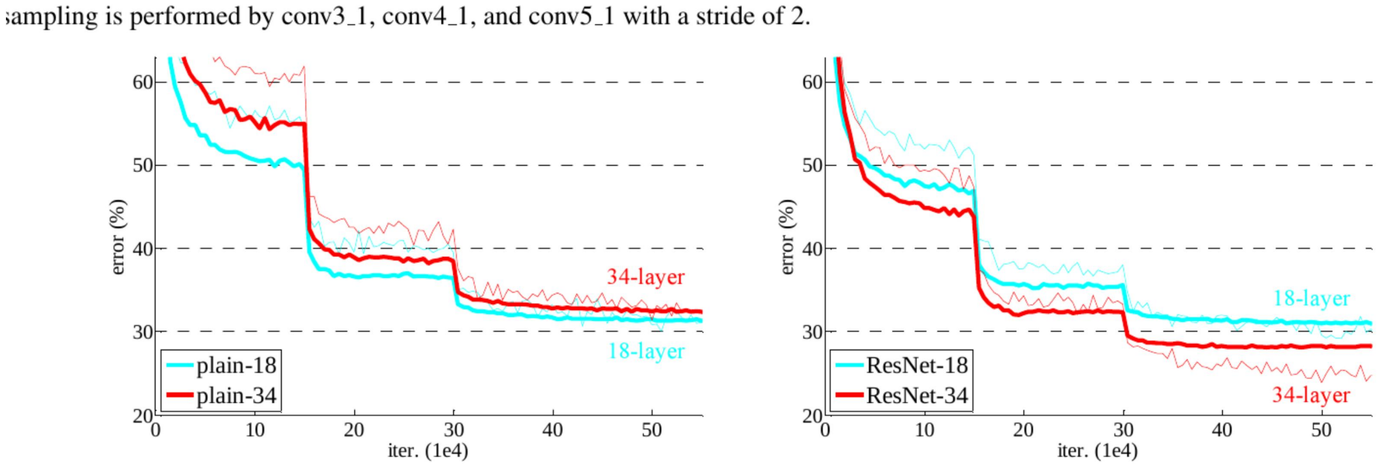
Covering the face lowers the class probability.

An intervention: test what the classifier relies on.

ResNet: making deeper image models train

Remember that "vanilla" deep networks fit worse.

Residual connections fix that! Combining convolutional layers + residual connections:



He et al., Deep Residual Learning for Image Recognition (2016) — Fig. 4. Thin lines: training error. Thick lines: validation error. Plain (left), residual (right).

ResNet 34-layer network fits better than 18-layer.

The "shortcuts" make the extra depth easier to use.

Up to 152 layers on ImageNet.

Architecture affects which weights the optimizer is able to find!

A modern convolutional network: ConvNeXt

ResNet helps us train deeper models.

ConvNeXt: update the convolutional block.

Larger spatial filters. LayerNorm.

GELU: a smooth alternative to ReLU.

Depthwise: one spatial filter per channel.

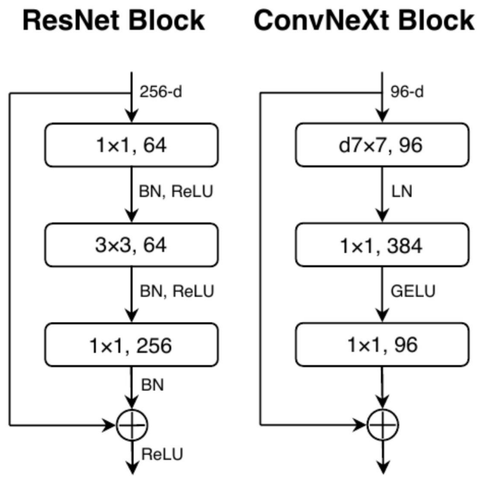
1×1 filters mix channels at each position.

Still convolutions + residual connections.

Vision transformers: image models using attention.

We'll study attention next lecture.

Same training techniques, different architectures.



Liu et al., A ConvNet for the 2020s (2022) — Fig. 4, ResNet and ConvNeXt blocks. BN: BatchNorm; LN: LayerNorm. Arrows are residual shortcuts.

Generating images

Can we learn without class labels?

So far: image $\rightarrow$ class, $p_\theta(y \mid x)$.

Classification needs labels.

What about unlabeled images?

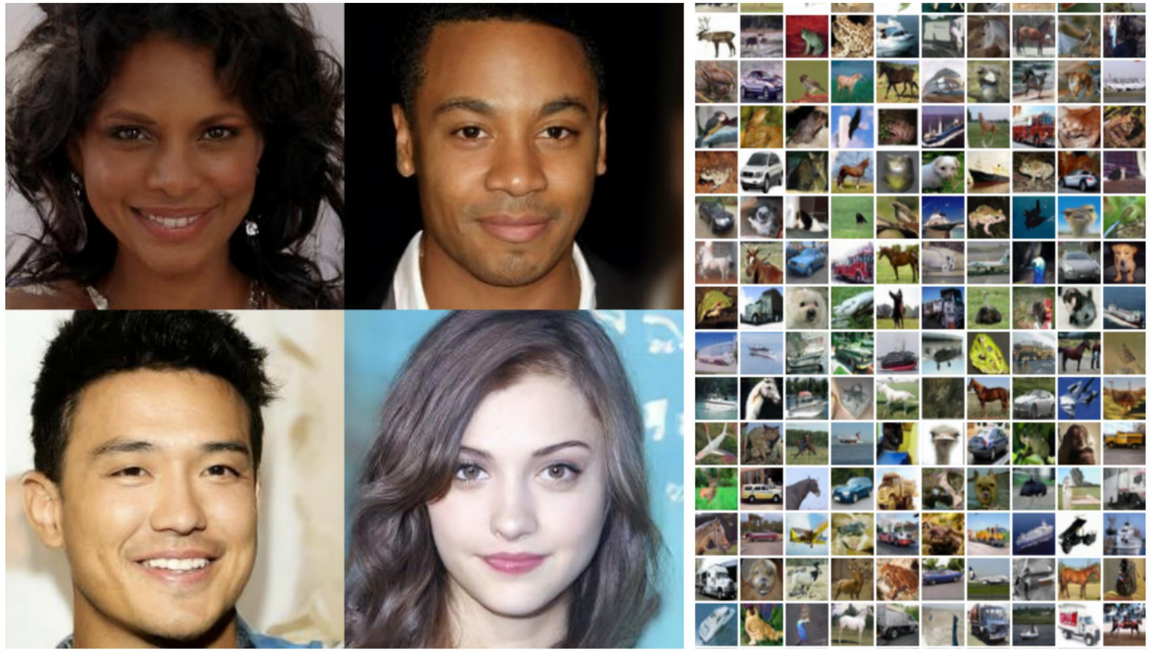
Q: What could we learn from the images themselves?

Generative model: model the distribution $p_\theta(x)$.

Then we could sample new images $x' \sim p_\theta(x)$.

One option: train a **diffusion model**.

Idea: learn to transform a "noisy" image into a real image.



Ho, Jain & Abbeel, Denoising Diffusion Probabilistic Models (2020) — Fig. 1. Generated samples, not training inputs.

Diffusion: prediction task

Our model has to predict some y from x. Which y?

We'll create a synthetic (x, y) pair using a clean image x_0:

- Choose noise level t.
- Draw Gaussian noise $\epsilon \sim \mathcal{N}(0, I)$.
- Mix image and noise:

$$x_t = \sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1 - \bar{\alpha}_t}\, \epsilon$$

$\bar{\alpha}_t$: fraction of signal remaining. Decreases with t.

Input: (x_t, t). Target: added noise ϵ.

$$\min_\theta \; \mathbb{E}_{x_0, t, \epsilon}\left[\|\epsilon - \epsilon_\theta(x_t, t)\|^2\right]$$

Train the same predictor at many noise levels.

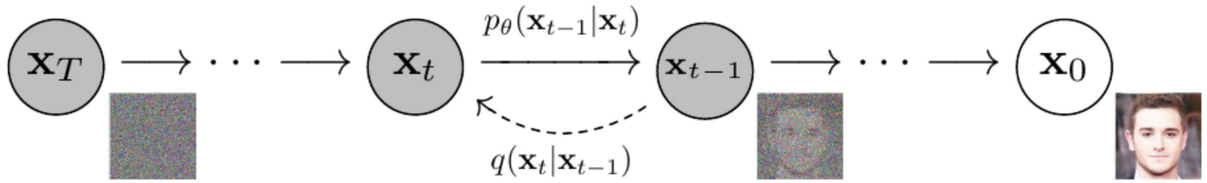
> 💡 **Insight**
>
> We generated the noise during training, so we know the target.
>
> **Self-supervised learning:** create targets from the data.

Ho, Jain & Abbeel, Denoising Diffusion Probabilistic Models (2020), simplified noise-prediction objective (Eq. 14).

Diffusion: generating an image

Now we can use the trained noise predictor to generate an image.

Start with random noise x_T. Take a small denoising step.



Ho et al., Denoising Diffusion Probabilistic Models (2020) — Fig. 2. The known corruption process q adds noise (right to left); the learned reverse process pθ removes it (left to right).

Repeat at smaller noise levels:

$x_T \to x_{T-1} \to \cdots \to x_0$, a generated image.



Ho et al., Denoising Diffusion Probabilistic Models (2020) — Fig. 6. These are predicted clean images at successive stages, rather than the noisy states themselves.

Extension: denoise using the noisy image *and* a text representation.

Same reverse process, now conditioned on the description.

Generalization

Back to generalization

We can train classifiers and generators.

But how well do they work on unseen examples?

> 🧠 **Recall**
>
> Lecture 9: finite-class bound depends on $\log |\mathcal{F}|$.
>
> $\mathcal{F}$: set of possible predictors.

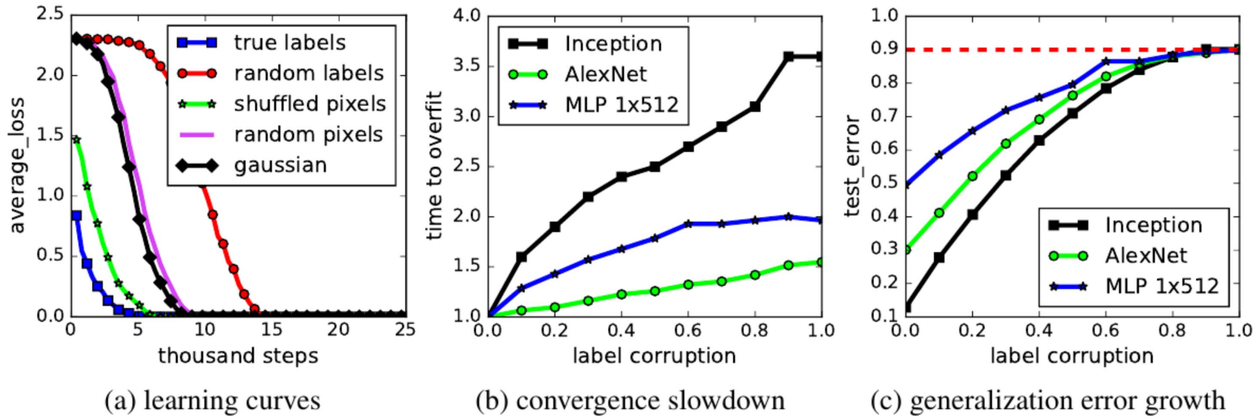
Neural networks seem to generalize very well! Maybe it's a small $\mathcal{H}$ somehow?

Test: can deep neural nets learn to predict arbitrary labels?

Experiment: use the images, but replace true labels with random ones.

What happens in training?

They can fit random labels too



(a) learning curves (b) convergence slowdown (c) generalization error growth

Zhang et al., Understanding Deep Learning Requires Rethinking Generalization (2017) — Fig. 1, original version. Training loss (left), time to fit (middle), test error (right).

They fit random labels too! Training error goes to 0.

Enough capacity to memorize the training set.

Meaningful labels: can generalize.

Random labels: no useful relationship for new examples.

Why does training choose a useful function on real data?

What if we keep increasing model size?

Recall the trade-off:

Larger class $\Rightarrow$ less approximation error, potentially more estimation error.

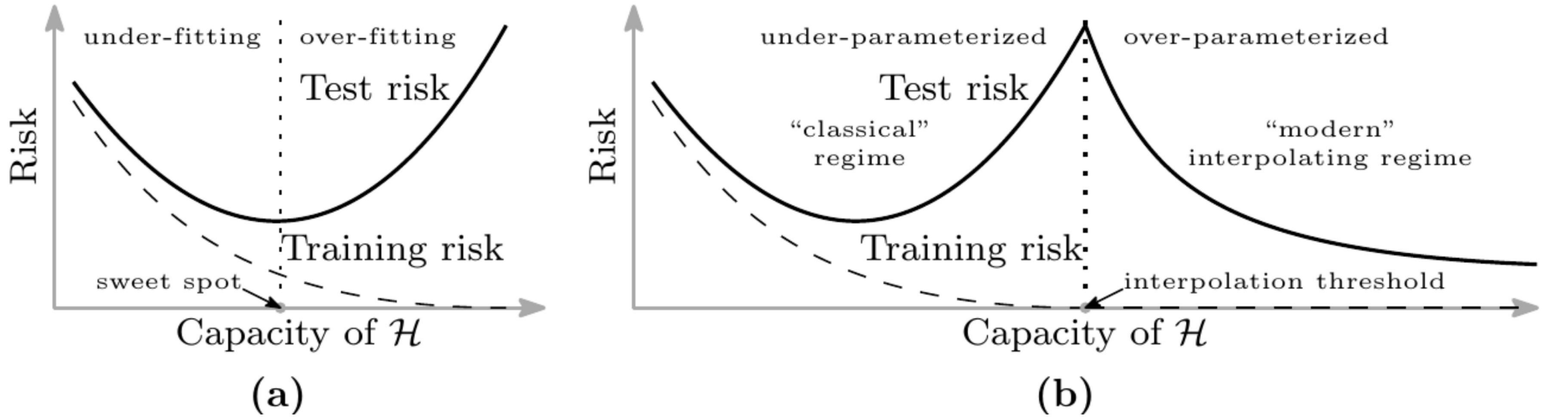
If we increase parameters and plot test error (after convergence), what do we see?

Initially, test error goes down (more expressive models), until...

Interpolation threshold: first model size that can fit all training examples.

Overfits badly! Like a polynomial function of degree n over n inputs.

What if we keep going?



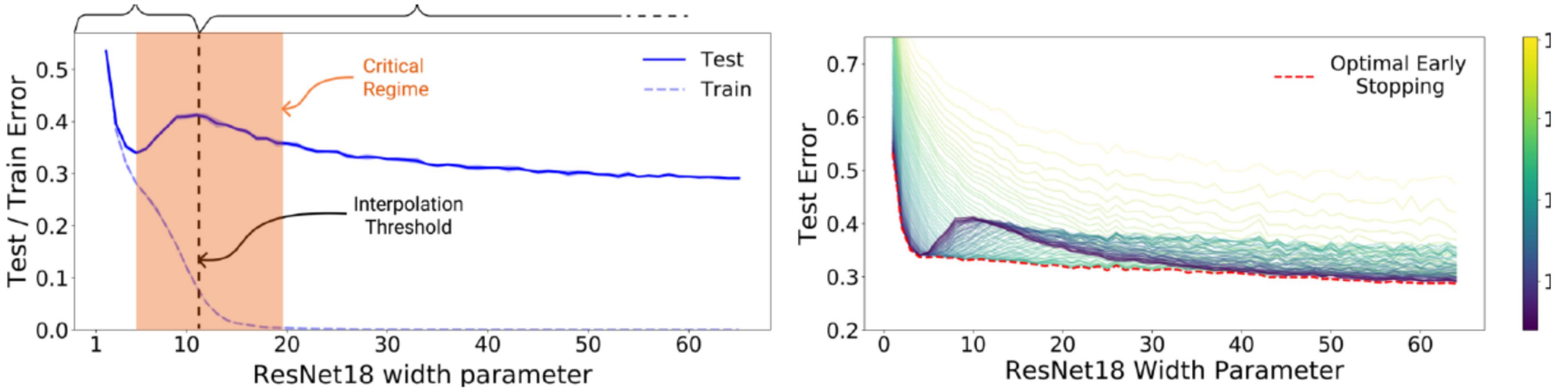
Belkin et al., Reconciling Modern Machine Learning Practice and the Bias–Variance Trade-off (2019) — Fig. 1. Schematic.

Beyond the threshold: test error falls again!

This is **double descent**. This phenomenon is recent! (it also happens in the linear case, even!)

Q: All these larger models fit. Why different test errors?

Double descent in deep networks



Nakkiran et al., Deep Double Descent: Where Bigger Models and More Data Hurt (2020) — Fig. 1. Left: ResNet18 width sweep, CIFAR-10 with 15% label noise, Adam for 4000 epochs. Right: curves at different training epochs.

Vary ResNet width. Keep data fixed.

Test error peaks near zero training error.

Wider still: same perfect training fit, lower test error.

The curve depends on data, noise, regularization, and training time.

Which function do we learn?

Many functions fit the same training set.

They can disagree on new inputs (i.e., generalize differently).

> **? Question**
> Which one does training choose?

- **Architecture:** shared local filters, residual shortcuts.
- **Data + augmentation:** useful patterns and invariances.
- **Optimization:** initialization and updates favor some solutions.
- **Regularization:** changes which solutions we prefer.

Just counting parameters can hide many of these differences

Belkin et al. (2019): double descent also in linear models with fixed random features.

Summary

- **Train depth:** schedules, initialization, normalization, shortcuts.
- **Normalize:** BatchNorm across examples; LayerNorm within one example.
- **Regularize:** dropout, augmentation, weight decay.
- **Images:** shared local filters $\rightarrow$ larger-scale features.
- **Diffusion:** learn to predict noise $\rightarrow$ generate images.
- **Generalization:** models can have same training error but extrapolate differently

Next: language models. Then combine text + image representations.