Deep learning

Fraida Fund

Contents

In this lecture

- Deep neural networks
- Challenges and tricks

Math prerequisites for this lesson: None.

Recap

Last week: neural networks with one hidden layer

- Hidden layer learns feature representation
- Output layer learns classification/regression tasks

With the neural network, the "transformed" feature representation is *learned* instead of specified by the designer.

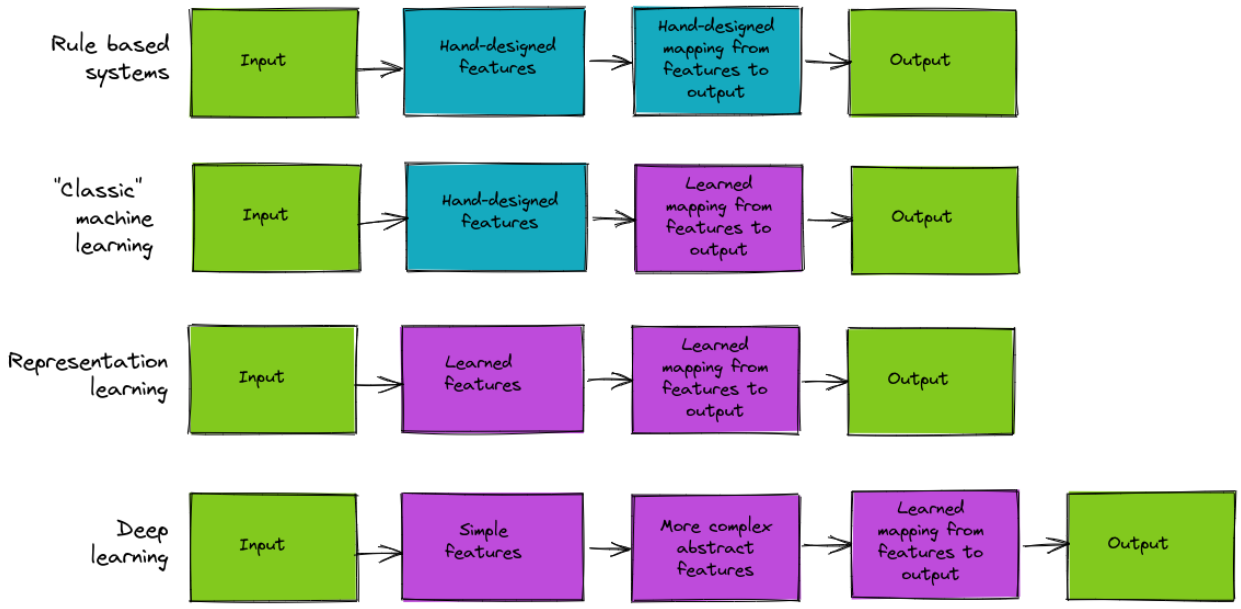

Figure 1: Image is based on a figure in Deep learning, by Goodfellow, Bengio, Courville.

A neural network with non-linear activation, with one hidden layer and many units in that layer *can* approximate virtually any continuous real-valued function, with the right weights. (Refer to the *Universal Approximation Theorem*.) But (1) it may need a very large number of units to represent the function, and (2) those weights might not be learned by gradient descent - the loss surface is very unfriendly.

Instead of a single hidden layer, if we use multiple hidden layers they can "compose" functions learned by the previous layers into more complex functions - use fewer units, and tends to learn better weights .

Deep neural networks

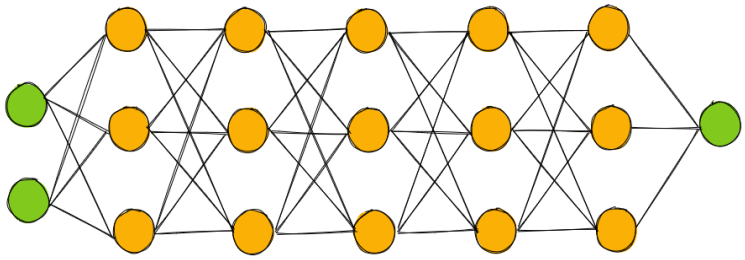

Figure 2: Illustration of a deep network, with multiple hidden layers.

Some comments:

- each layer is fully connected to the next layer
- each unit still works the same way: take the weighted sum of inputs, apply an activation function, and that's the unit output
- still trained by backpropagation

We call the number of layers the "depth" of the network and the number of hidden units in a layer its "width."

Double descent: animation

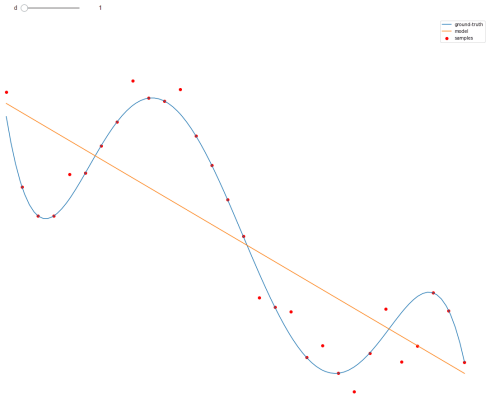

Figure 3: Polynomial model before and after the interpolation threshold. Image source: Boaz Barak, click link to see animation.

Explanation (via Boaz Barak):

> When d of the model is less than d_t of the polynomial, we are "under-fitting" and will not get good performance. As d increases between d_t and n, we fit more and more of the noise, until for $d = n$ we have a perfect interpolating polynomial that will have perfect training but very poor test performance. When d grows beyond n, more than one polynomial can fit the data, and (under certain conditions) SGD will select the minimal norm one, which will make the interpolation smoother and smoother and actually result in better performance.

Double descent curve

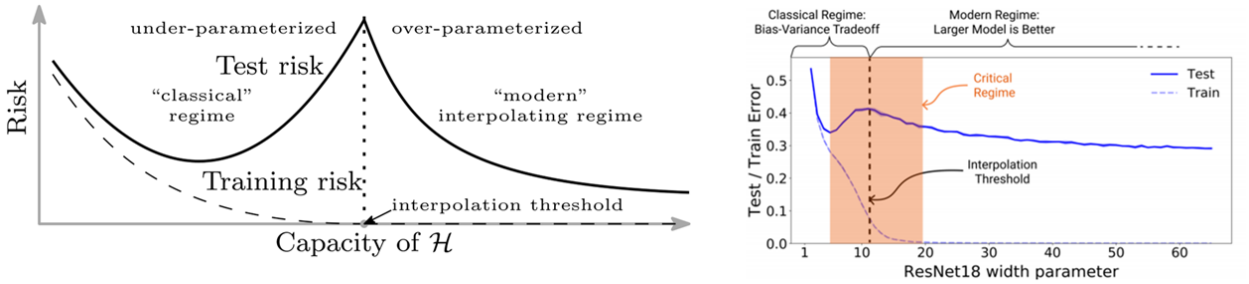

Figure 4: Double descent curve (left) and realization in a real neural network (right).

With a deep neural network, we are not trying to operate in the "classical ML" bias-variance tradeoff regime (to the left of the interpolation threshold).

(Interpolation threshold: where the model is just big enough to fit the training data exactly.)

- too-small models: can't represent the "true" function well (lacks capacity to learn complicated data representations!)
- too-big models (before interpolation threshold): memorizes the input, doesn't generalize well to unseen data (very sensitive to noise)
- REALLY big models: many possible weights that memorize the input, our challenge is to find the weight combination that memorizes the input *and* does well on unseen data

This is complicated by the "loss landscape" of a deep neural network (trained using backpropagation over the computational graph) looking not so friendly...

Loss landscape

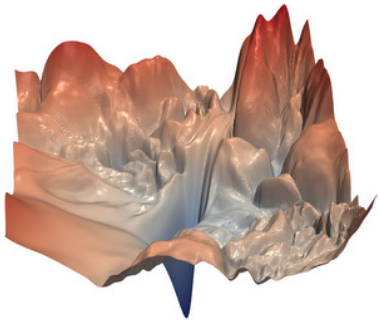

Figure 5: "Loss landscape" of a deep neural network in a "slice" of the high-dimensional feature space.

Image source: Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer and Tom Goldstein. Visualizing the Loss Landscape of Neural Nets. NIPS, 2018.

There are a variety of techniques we can use to improve the network's ability to learn good weights + find them efficiently, even on this difficult loss surface. (Including what we call "conditioning" techniques, and some broader techniques.)

Addressing the challenges

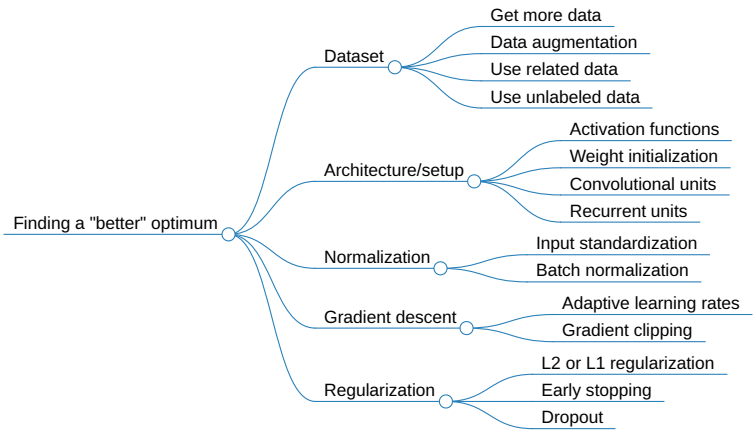

Figure 6: Image credit: Sebastian Raschka

Dataset

- Get more data
- **Data augmentation**
- **Use related data (transfer learning)**
- Use unlabeled data (self-supervised, semi-supervised)

We won't talk much about getting more data, but if it's possible to get more labeled data, it is almost always the most helpful thing you can do!

- Example: JFT-300M, a Google internal dataset for training image models, has 300M images
- Example: GTP-3 trained on 45TB of compressed plaintext, about 570GB after filtering

Reference: Revisiting Unreasonable Effectiveness of Data in Deep Learning Era. Chen Sun, Abhinav Shrivastava, Saurabh Singh, Abhinav Gupta; Proceedings of the IEEE International Conference on Computer Vision (ICCV), 2017, pp. 843-852.

It's not always possible to get a lot of *labeled* data for training a supervised learning model. But sometimes we can use *unlabeled* data, which is much easier to get. For example:

- In self-supervised learning, the label can be inferred automatically from the unlabeled data. e.g. GPT is trained to predict "next word".
- In semi-supervised/weakly-supervised learning learining, we generate labels (probably imperfectly) for unlabeled data (maybe using a smaller volume of labeled data to train a model to label the data!)

These are mostly out of scope of this course. But we *will* talk about data augmentation and transfer learning...

Data augmentation

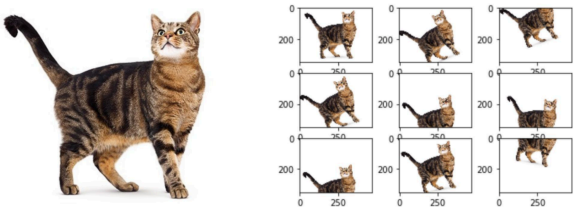

Figure 7: Data augmentation on a cat image.

It doesn't restrict network capacity - but it helps generalization by increasing the size of your training set!

- For image data: apply rotation, crops, scales, change contrast, brightness, color.
- For text you can replace words with synonyms, simulate typos.
- For audio you can adjust pitch or speed, add background noise etc.

Transfer learning

Idea: leverage model trained on *related* data.

State-of-the-art networks involve millions of parameters, huge datasets, and days of training on GPU clusters

But you don't have to repeat this process "from scratch" each time you train a neural network. (For many tasks, you may not even have enough data to get good results by training a network "from scratch".)

The "feature extraction" part of a neural network trained on related data, is likely still very useful for a slightly different task.

Transfer learning from pre-trained networks

Use pre-trained network for a different task

- Use early layers from pre-trained network, freeze their parameters
- Only train small number of parameters at the end

The base model is a powerful feature extractor (learns transformation into a more useful feature space), then you just have to train for the mapping from this feature space to the target variable.

In practice: when applying deep learning, we almost always use a pre-trained base model! It saves time, energy, and cost.

Example: To pre-train the 7B parameter (smallest) Llama 2, Meta's open source language model, takes 184,320 GPU hours (21 GPU YEARS!) and anywhere from $100,000-800,000 (depending on cost of GPU instances).

Transfer learning illustration (1)

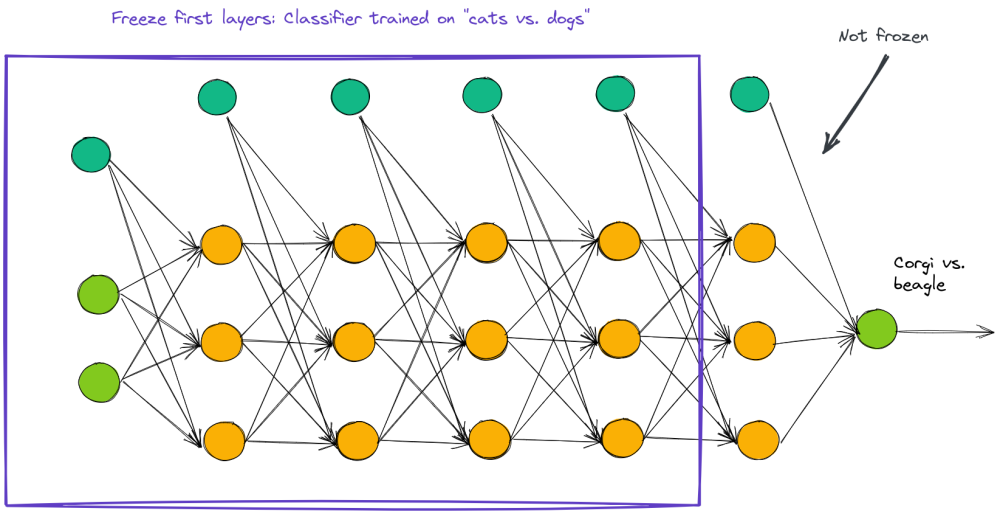

Figure 8: When the network is trained on a very similar task, even the abstract high-level features are probably very relevant, so you might tune just the classification head.

Transfer learning illustration (2)

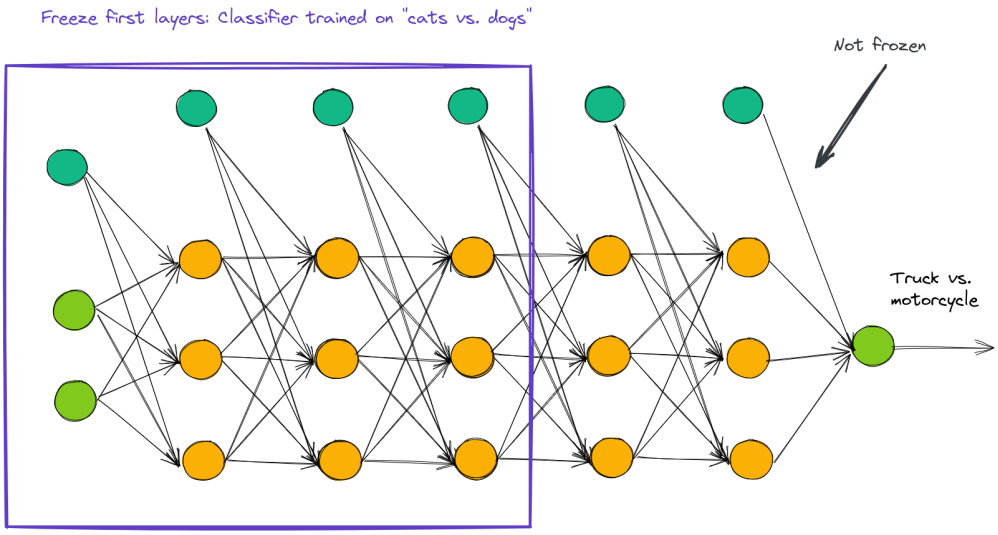

Figure 9: If the original network is not as relevant, may fine-tune more layers.

Architecture/setup

- **Activation functions** (+ skip connections)
- **Weight initialization**
- **Convolutional units**
- Recurrent units
- ... many more ideas

Again, this is mostly out of scope of this course, but we'll talk about some of these items very briefly.

Recall: activation functions

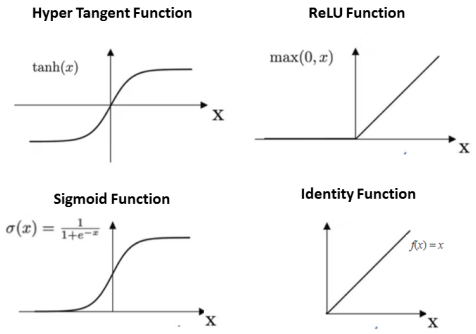

Figure 10: Candidate activation functions for a neural network.

Sigmoid: vanishing gradient (1)

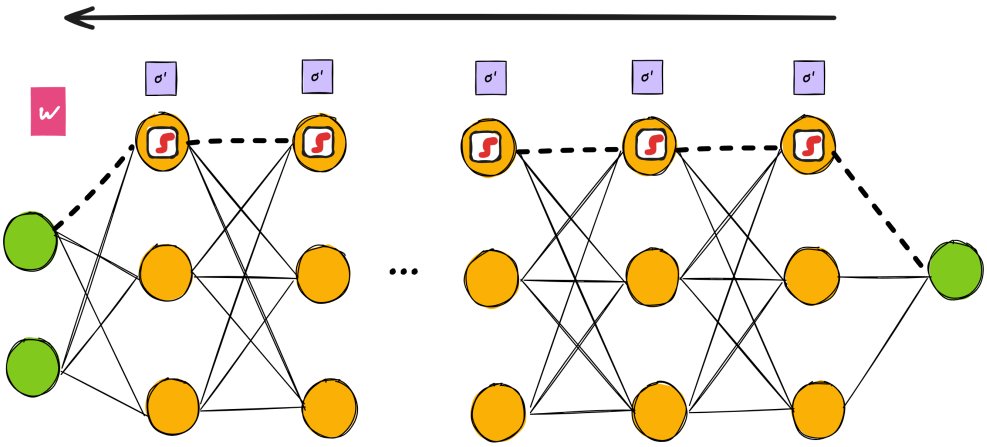

Figure 11: Computing gradients by backpropagation. At a hidden unit, $\delta_j = g'(z_j) \sum_k w_{k,j}\delta_k$.

Suppose we want to compute the gradient of the loss with respect to the weight shown in pink.

- We will *multiply* local gradients (pink) all along each path between weight and loss function, starting from the end and moving toward the input.
- Then we *add* up the products of all the paths.
- With σ activation, gradient along each path includes the product of a LOT of σ' terms.

Sigmoid: vanishing gradient (2)

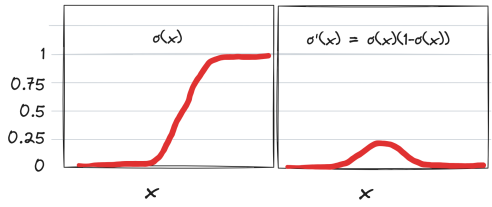

Figure 12: Sigmoid function and its derivative.

What happens when you are in the far left or far right part of the sigmoid?

- Gradient is close to zero
- Weight updates are also close to zero
- The "downstream" gradients will also be values close to zero! (Because of backpropagation.)
- And, when you multiply quantities close to zero - they get even smaller.

The network "learns fastest" when the gradient is large. When the sigmoid "saturates", it "kills" the neuron!

Even the maximum value of the gradient is only 0.25 - so the gradient is always less than 1, and we know what happens if you multiply many quantities less than 1...

(tanh is slightly better - gradient has a larger max + some other advantages - still has vanishing gradient.)

ReLU: Dead ReLU

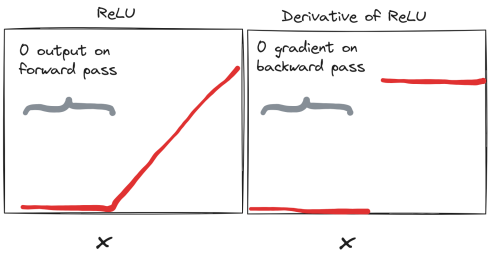

Figure 13: ReLU function and its derivative.

ReLU is a much better non-linear function:

- does not saturate in positive region
- very very fast to compute

But, can "die" in the negative region. Once we end up there (e.g. by learning negative bias weight) we cannot recover - since gradient is also zero, gradient descent will not update the weights.

(ReLU is also more subject to "exploding" gradient than sigmoid/tanh.)

ReLU: Leaky ReLU

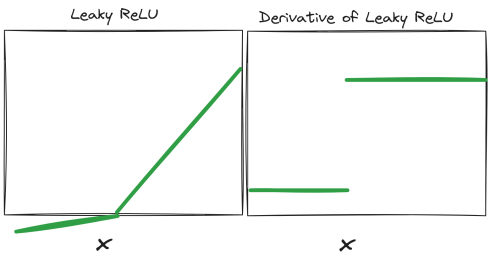

Figure 14: Leaky ReLU function and its derivative.

When input is less than 0, the ReLU (and downstream units) is *completely* dead (not only very small!)

Alternative: **leaky ReLU** has small (non-zero) gradient α in the negative region - can recover.

$$f(x) = \max(\alpha x, x)$$

Skip connections

Alternative solution for "vanishing gradient":

- Direct connection between some higher layers and lower layers
- A "highway" for gradient info to go directly back to lower layers

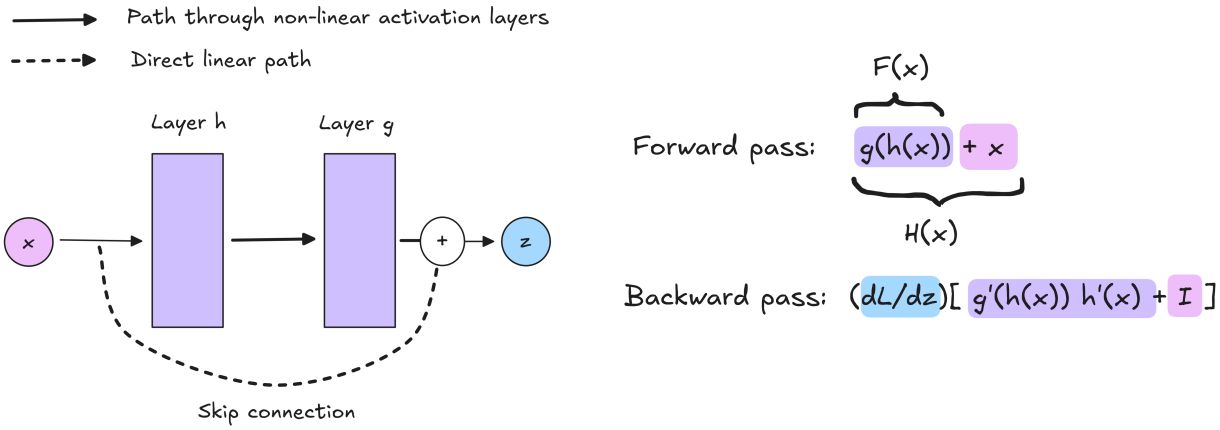

Figure 15: Diagram of skip connection.

Suppose we want this sequence of layers to learn $z = H(x)$.

Instead of learning $H(x)$ directly, the network learns the **residual function**:

$$\mathcal{F}(x) = H(x) - x$$

Then it reconstructs the desired transformation as:

$$H(x) = \mathcal{F}(x) + x$$

This helps avoid vanishing gradient.

In the standard neural network, we would have:

$$\frac{dL}{dx} = \frac{dL}{dz} \cdot g'(h(x)) \cdot h'(x)$$

where $g'()$ and $h'()$ multiply the gradient that backpropagates from the higher layers, $\frac{dL}{dz}$, and scale it down. But with the skip connection, we now have

$$\frac{dL}{dx} = \frac{dL}{dz} \cdot \left(\frac{d\mathcal{F}(x)}{dx} + I \right) = \frac{dL}{dz} \cdot (g'(h(x)) \cdot h'(x) + I)$$

The identity matrix I passes on the gradient from higher layers as an additive term, without scaling it down.

Weight initialization

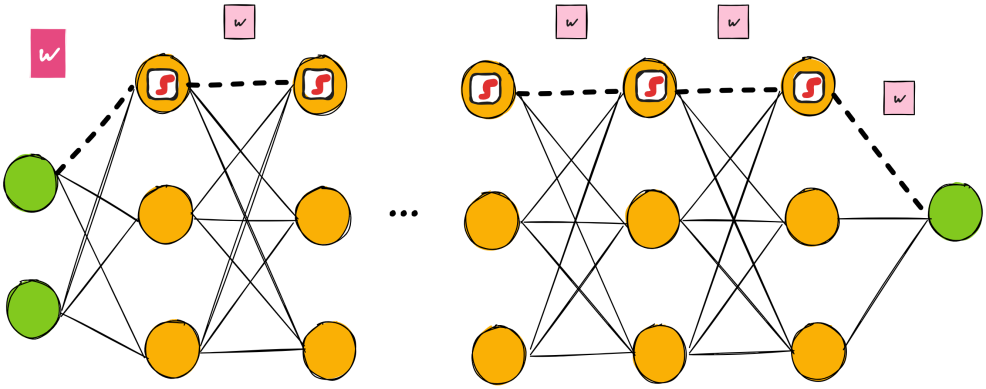

Figure 16: One path in forward pass on a neural network.

What if we initialize weights to:

- **zero?** If weights are all initialized to zero, all the outputs are zero (for any input) - won't learn.
- **a constant (non-zero)?** If weights are all initialized to the same constant, we are more prone to "herding" - hidden units all move in the same direction at once, instead of "specializing".
- **a normal random value with small σ?** Small normal random values work well for "shallow" networks, but not for deep networks - it makes the outputs "collapse" toward zero at later layers.
- **a normal random value with large σ?** Large normal random values are bad - it makes the outputs "explode" at later layers.

Weight initialization - normal

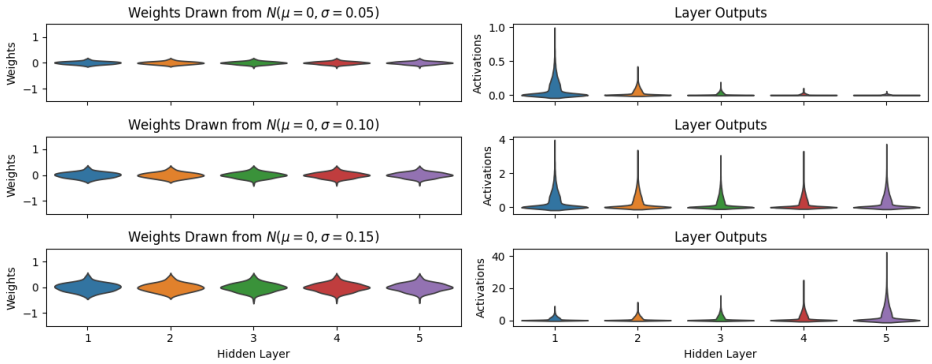

Figure 17: Initial weights and ReLU unit outputs for each layer in a network.

- top row: too-small initial weights, by the last layer the outputs "collapse" toward zero
- middle row: good initial weights, distribution is similar from input to output
- bottom row: too-large initial weights, by the last layer the outputs "explode"

Desirable properties for initial weights - principle

- The mean of the intial weights should be around 0
- The variance of the activations should stay the same across every layer

If you are interested, here's a derivation.

Desirable properties for initial weights - practice

- For tanh: Xavier scales by $\frac{1}{\sqrt{N_{in}}}$
- For ReLU: He scales by $\frac{2}{\sqrt{N_{in}}}$

N_{in} is the number of inputs to the layer ("fan-in").

Weight initialization - He

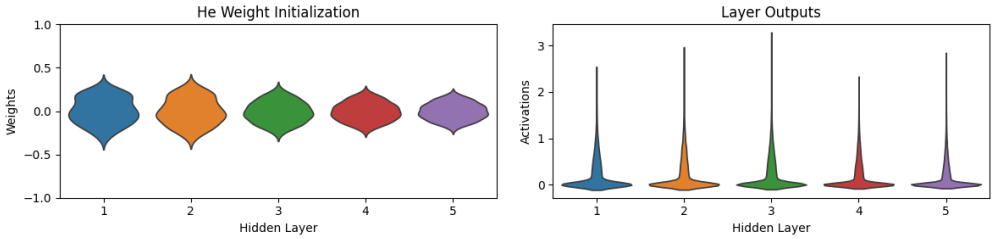

Figure 18: Initial weights and ReLU unit outpus for each layer in a network, He initialization. In this example, the size of the layers is: 100, 150, 200, 250, 300.

Convolutional units

Problems with "fully connected" layers for images

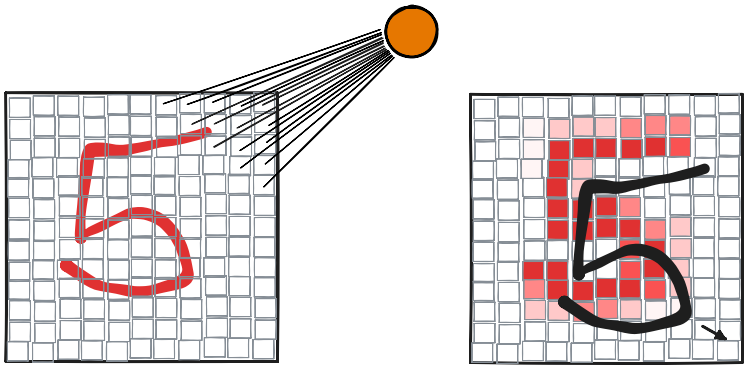

Figure 19: Problems with fully connected layers.

Convolutional neural networks address two major problems that make it difficult to train a "fully connected" neural network:

1. Each pixel is a feature, images tend to be very large, so a "fully connected" layer requires a very large number of parameters. (Each "neuron" in the first layer requires a weight for every pixel in the image!)
2. Each learned weight corresponds to specific pixels in the imgae. If the relevant pixels are at a different position in the image, the weights that were learned are not helpful for that other position. (i.e. it is not *spatial translation invariant*.)

Using convolution to address the problem

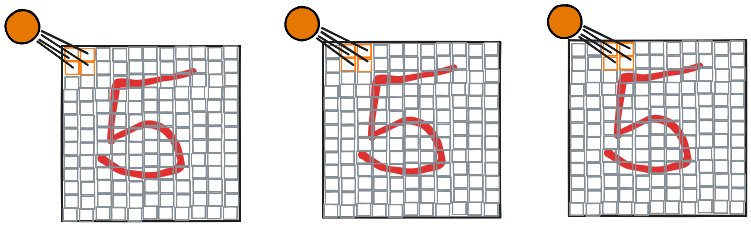

Figure 20: Addressing the problem.

With a convolutional layer (not "fully connected"):

1. The unit connects to one "patch" of the image at a time, so it only needs as many weights as there are pixels in the patch. The same weights are *shared* as the unit moves across the image, one patch at a time.
2. It can "match" a specific arrangement of pixels *anywhere* it occurs in the image, not only one specific location.

The convolution operation - one “patch”

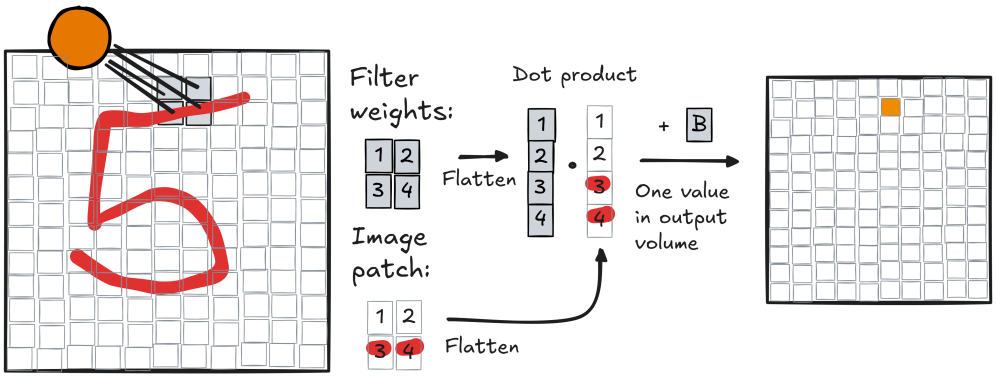

Figure 21: Convolution on one patch.

- Layer has a set of learnable “filters” (illustration shows one filter)
- Each filter has small width and height, but full depth
- During forward pass, filter “slides” across width and height of input, and computes dot product
- Effectively performs “convolution”

The convolution operation - stride and padding

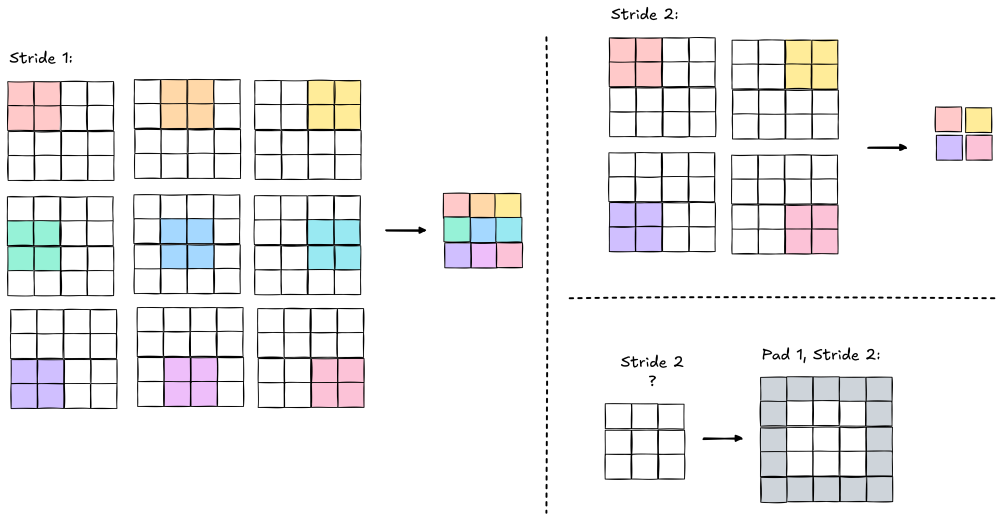

Figure 22: Stride and padding.

The convolution operation - full depth

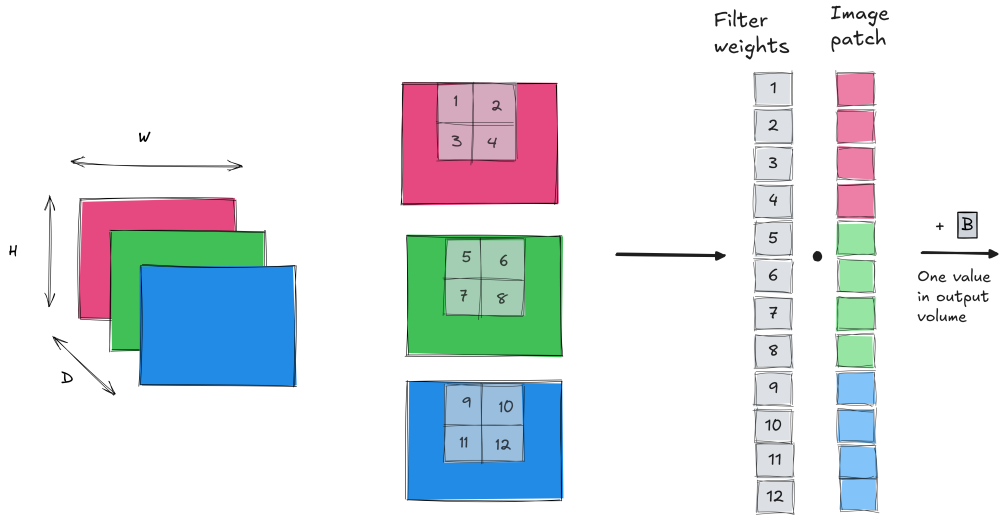

Figure 23: Convolution across full depth.

The convolution operation - multiple filters

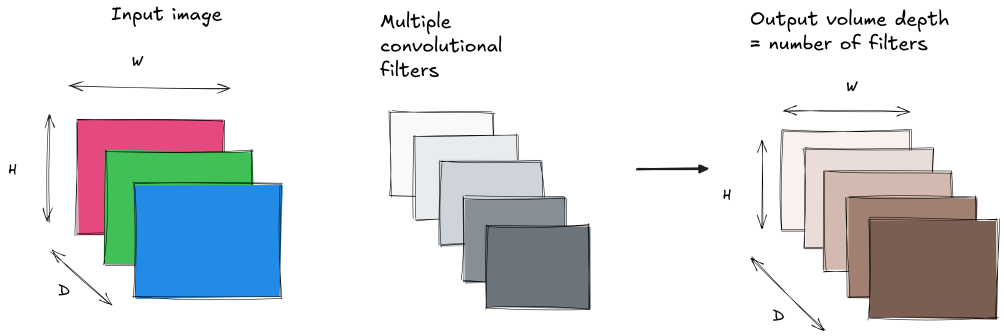

Figure 24: Convolution with multiple filters.

Note that the output is a 3D volume - can be input to another conv layer!

Basic dimension arithmetic:

- Accepts input volume $W_1 \times H_1 \times D_1$
- Four hyperparameters: number of filters K, filter size F, stride S, amount of zero padding P
- Produces volume of size

$$W_2 = \frac{W_1 - F + 2P}{S} + 1, H_2 = \frac{H_1 - F + 2P}{S} + 1$$
$$D_2 = K$$

- With parameter sharing: $F \cdot F \cdot D_1$ weights per filter, for $F \cdot F \cdot D_1 \cdot K$ weights and K biases

The convolution operation - all together

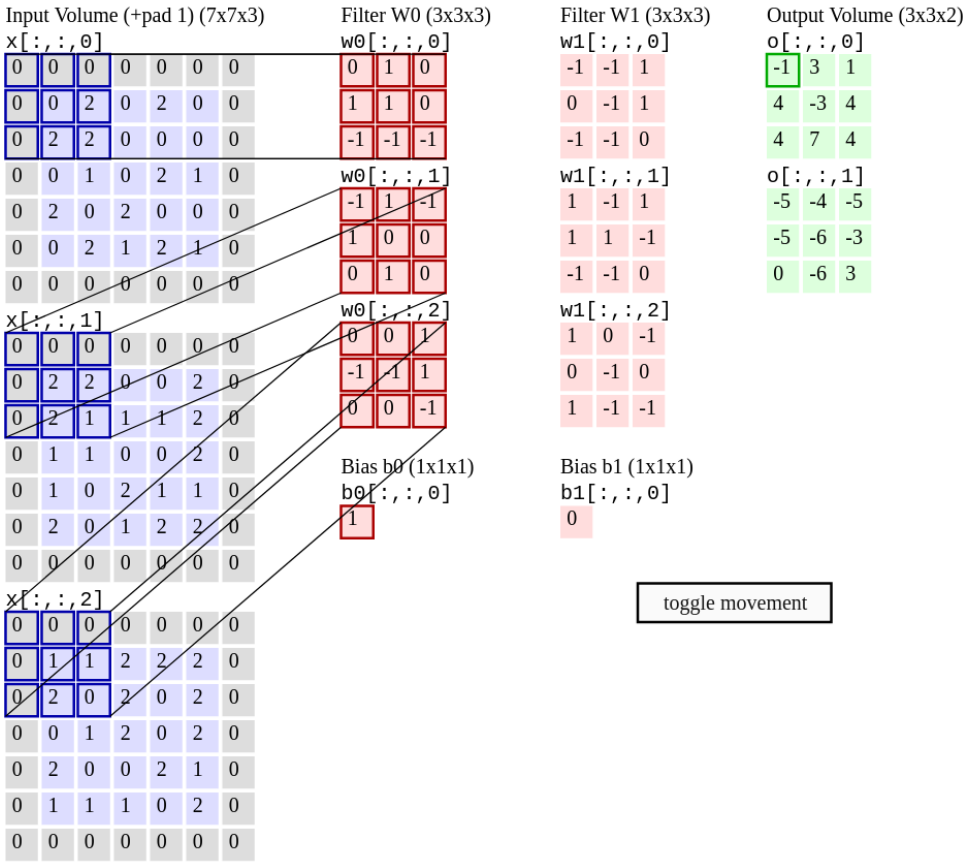

Figure 25: Animated demo at https://cs231n.github.io/assets/conv-demo/index.html

Basic insight - parameter sharing:

- A particular filter with a set of weights represents a feature to look for
- If it is useful to look for a feature at position x, y, it is probably useful to look for the same feature at x', y'
- All neurons within a "depth slice" can share the same weights.

Activation

- Convolutional typically followed by non-linear activation function e.g. ReLU
- Several Conv + ReLU layers may be followed by a *pooling* layer

Pooling layer

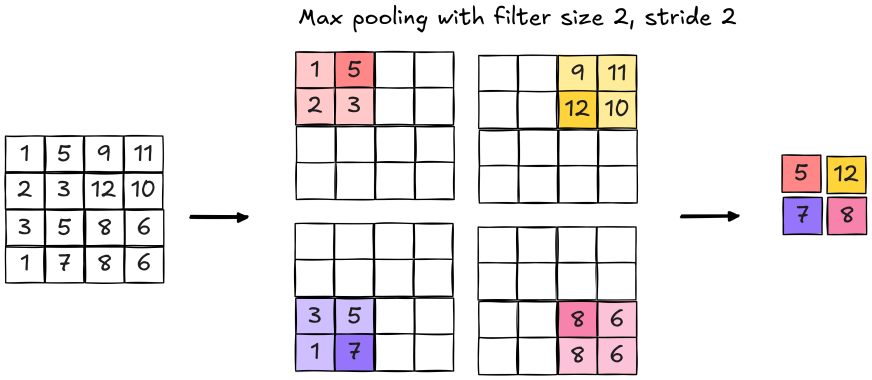

Figure 26: Example of pooling layer.

- No parameters! Just applies an aggregating function
- Typically uses max operation (other possible operations: mean, median)
- Reduces spatial size of image (reduce computation, prevent overfitting)
- Typical example: 2x2 filter size, stride of 2, downsamples by a factor of 2 along width and height
- Works independently on each depth slice

Pooling math:

- Accepts input volume $W_1 \times H_1 \times D_1$
- Two hyperparameters: filter size F, stride S
- Produces volume of size

$$W_2 = \frac{W_1 - F}{S} + 1, H_2 = \frac{H_1 - F}{S} + 1, D_2 = D_1$$

The typical "LeNet"-like architecture

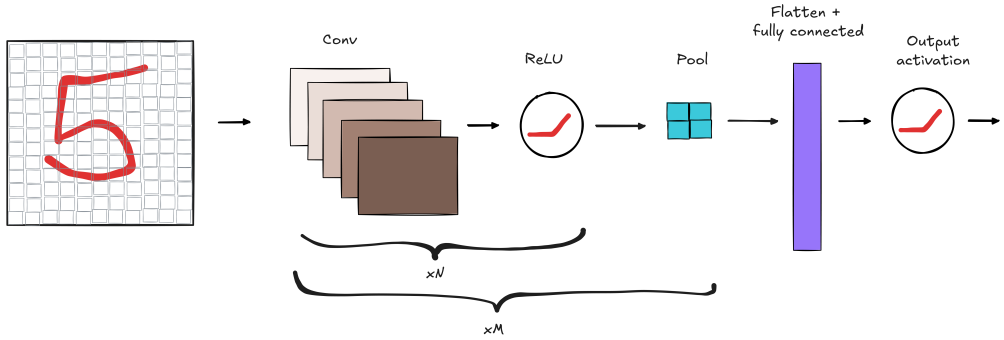

Figure 27: "LeNet"-like architecture.

Actual LeNet-5 (1998)

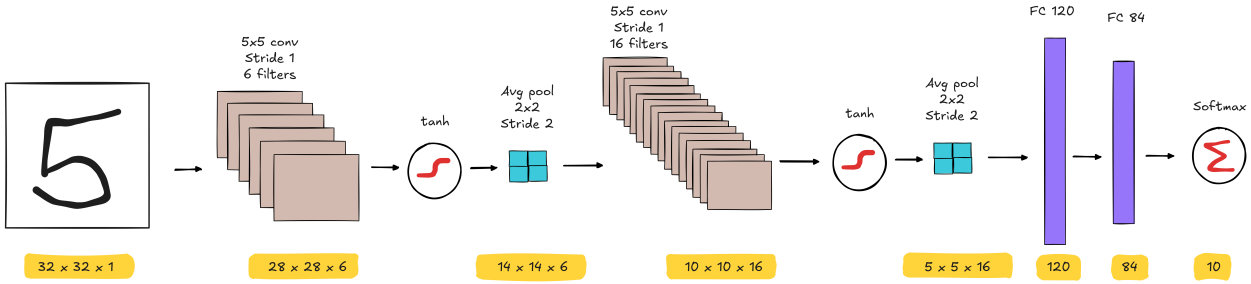

Figure 28: LeNet-5.

Recurrent neural networks

- Where ConvNet find *spatial* patterns wherever they occur in images,
- RNNs find *temporal patterns* wherever they occur in a sequence

Normalization

- **Input standardization**
- **Batch normalization**

Data pre-processing

You can make the loss surface much "nicer" by pre-processing:

- Remove mean (zero center)
- Normalize (divide by standard deviation)
- OR decorrelation (whitening/rotation)

There are several reasons why this helps. We already discussed the "ravine" in the loss function that is created by correlated features.

Data preprocessing (1)

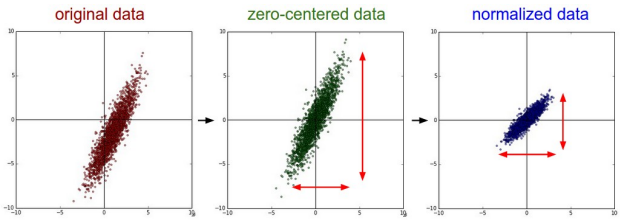

Figure 29: Image source: Stanford CS231n.

Data preprocessing (2)

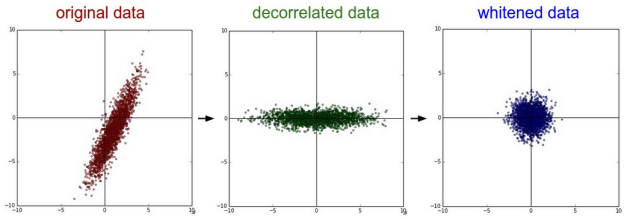

Figure 30: Image source: Stanford CS231n.

Input standardization helps with the first hidden layer, but what about the intermediate hidden layers?

Batch normalization

- Re-center and re-scale between layers
- Training: Mean and standard deviation per training mini-batch
- Test: Using fixed statistics

Gradient descent

Regularization

- **L2 or L1 regularization**
- **Early stopping**
- **Dropout**

L1 or L2 regularization

As with other models, we can add a penalty on the norm of the weights.

Normal gradient descent update rule:

$$w_{i,j}^{t+1} = w_{i,j}^{t} - \alpha \frac{\partial L}{\partial w_{i,j}^{t}}$$

With L2 regularization:

$$w_{i,j}^{t+1} = w_{i,j}^{t} - \alpha (\frac{\partial L}{\partial w_{i,j}^{t}} + \frac{2\lambda}{n} w_{i,j}^{t})$$

Often called “weight decay” in the context of neural nets.

Early stopping

- Compute validation loss each performance
- Stop training when validation loss hasn’t improved in a while
- Risk of stopping *too* early

Important: *must* divide data into training, validation, and test data - use validation data (not test data!) to decide when to stop training.

Dropout

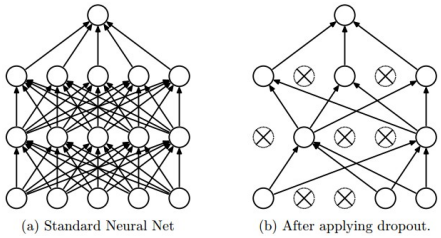

Figure 31: Dropout networks.

- During each training step: some portion of neurons are randomly “dropped”.
- During each test step: don’t “drop” any neurons, but we need to scale activations by dropout probability

Why does it work? Some ideas:

- Forces some redundancy, makes neurons learn robust representation
- Effectively training an ensemble of networks (with shared weights)

Note: when you use Dropout layers, you may notice that the validation/test loss seems better than the training loss! Why?

Big picture: What are all these techniques for?

- Allow us to train models with more ‘capacity’
- Improve performance even without adding ‘capacity’

Example: Deep Neural Nets: 33 years ago and 33 years from now

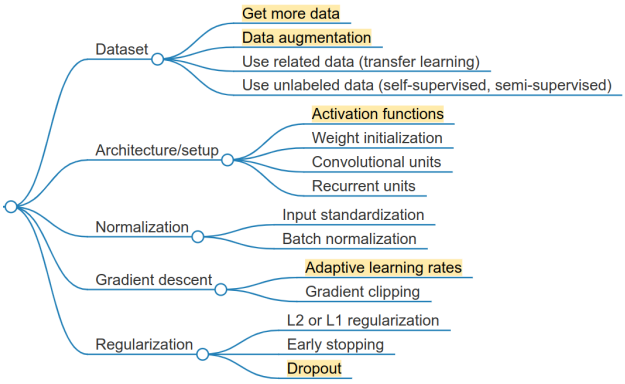

Figure 32: Techniques we will apply in this example.

In the Colab lesson, we will reproduce a 1989 paper about a neural network for handwritten digit classification, that was used in the late 90s to process 10-20% of all checks in the US.

- The original paper had 5% error, our realization has about 4.14%
- With a bunch of these changes (but keeping the basic network the same) we get to 2.09%
- The original model without any changes, but with more training data, gets to 3.05%
- With our changes + more data, we get to 1.31%

What does it mean that we can do this without changing the basic network? It means the network always had the *capacity* to do this well, but wasn't learning the best weights.

Example: Progress on ImageNet

- Data: 1.2M images from 1000 categories
- ImageNet Large Scale Visual Recognition Challenge (ILSVRC): running since 2010

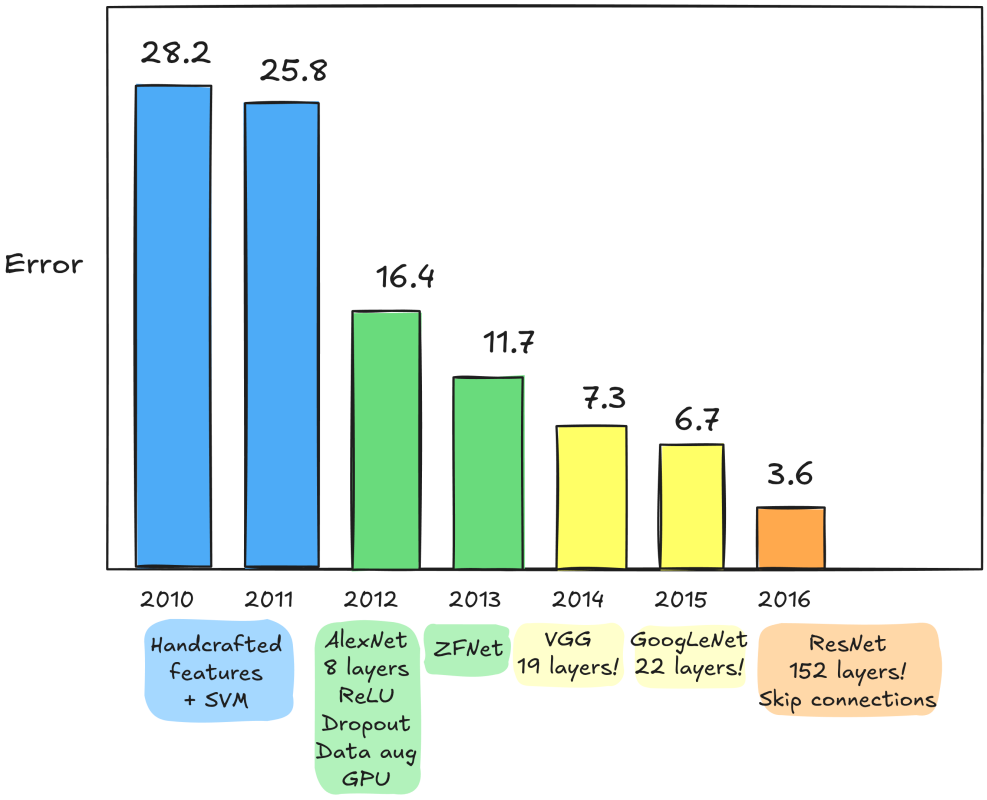

Figure 33: Progress on ImageNet.

If you want to learn more, including receptive field, 1x1 convolution, and ConvNet architectures, I recommend:

- Lecture 2A and Lecture 2B from FSDL 2021
- Chapter 7 and Chapter 8 of Dive into Deep Learning