Deep learning

Fraida Fund

Contents

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.

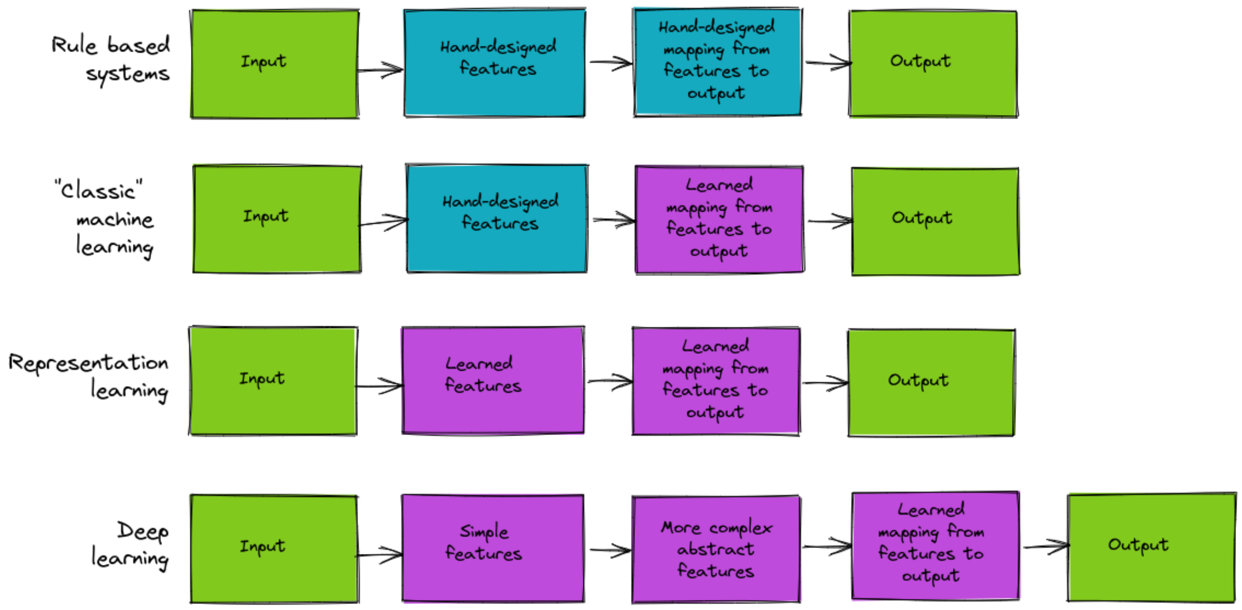
Rule based systems
Input
Hand-designed features
Hand-designed mapping from features to output
Output
"Classic" machine learning
Input
Hand-designed features
Learned mapping from features to output
Output
Representation learning
Input
Learned features
Learned mapping from features to output
Output
Deep learning
Input
Simple features
More complex abstract features
Learned mapping from features to output
Output

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 it *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

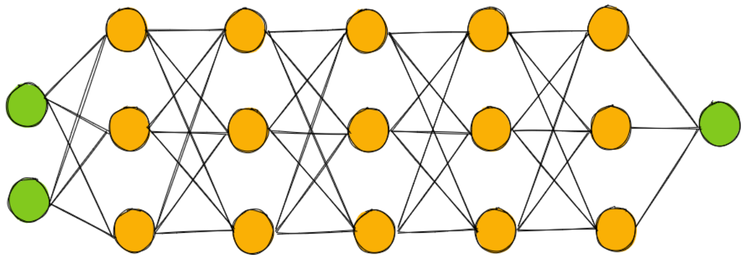

Figure 2: Illustration of a deep network, with multiple hidden layers.

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."

Challenges with deep neural networks (1)

- Efficient learning
- Generalization

Loss landscape

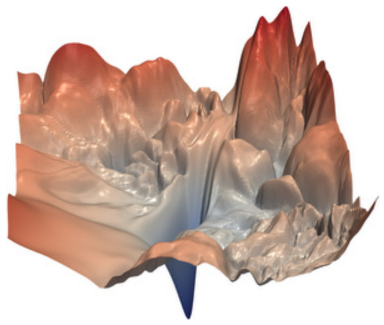

Figure 3: "Loss landscape" of a deep neural network in a "slice" of the high-dimensional feature space.

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

Neural networks are optimized using backpropagation over the computational graph, where the loss is a very challenging function of *all* the weights. (Not convex!)

Double descent curve

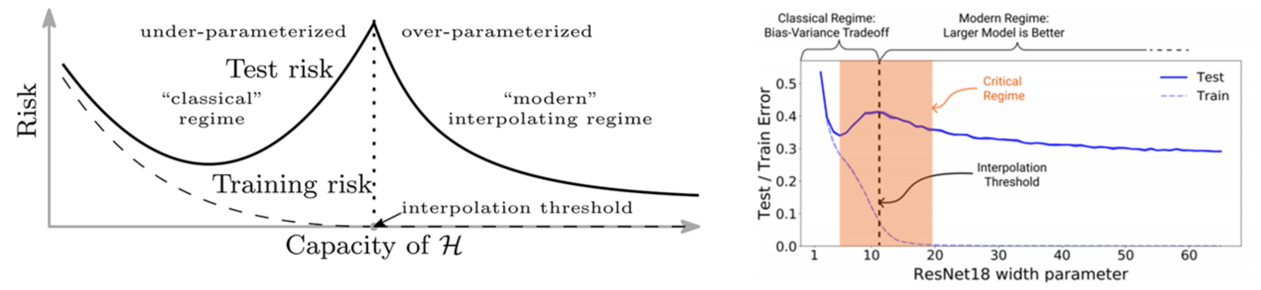

Figure 4: Double descent curve (left) and realization in a real neural network (right).

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
- 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, but SGD finds weight combination that memorizes the input *and* does well on unseen data

Double descent: animation

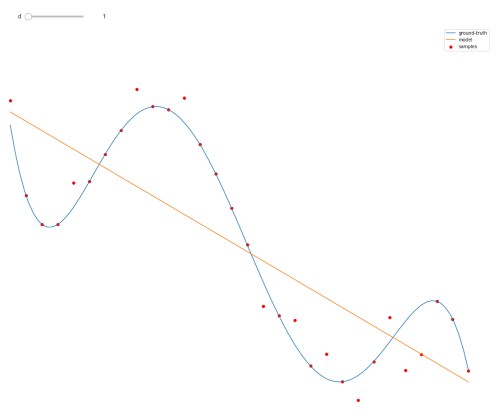

Figure 5: Polynomial model before and after the interpolation threshold. Image source: Boaz Barak, click link to see animation.

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.

What this means: in practice, we let the network get big (have capacity to learn complicated data representations!) and use other methods to help select a "good" set of weights from all these candidates.

Challenges with deep neural networks (2)

- Efficient learning
- Generalization

In deep learning, we don't want to use "smaller" (simpler) models, which won't be as capable of learning good feature representations. Instead, lots of work around (1) finding good weights quickly, and (2) finding weights that will generalize.

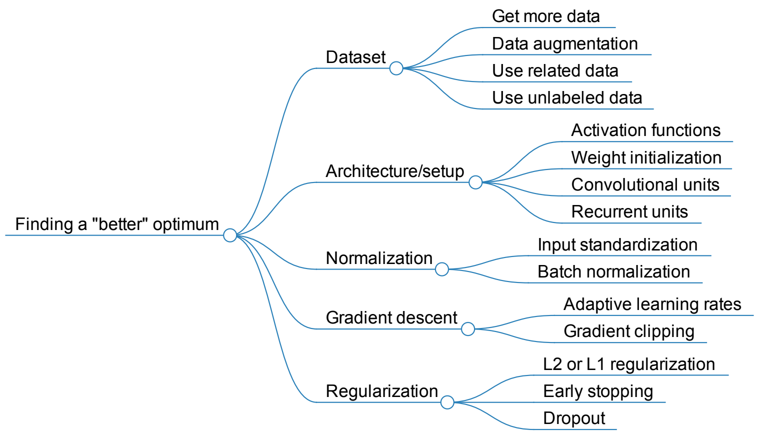

Figure 6: Image credit: Sebastian Raschka

Dataset

Data augmentation

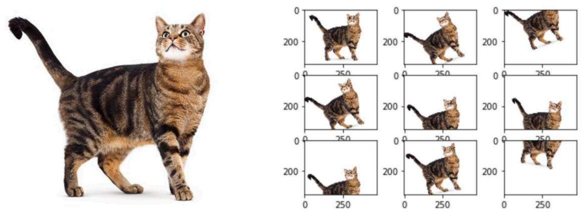

Figure 7: Data augmentation on a cat image.

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

Apply rotation, crops, scales, change contrast, brightness, color... etc. during training.

Transfer learning

Idea: leverage model trained on *related* data.

Using pre-trained networks

- State-of-the-art networks involve millions of parameters, huge datasets, and days of training on GPU clusters
- Idea: share pre-trained networks (network architecture and weights)
- Some famous networks for image classification: Inception, ResNet, and more
- Can be loaded directly in Keras

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

Transfer learning illustration (1)

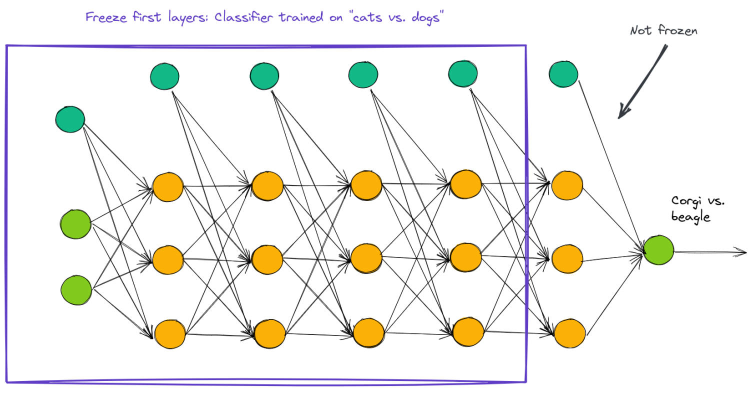

Figure 8: When the network is trained on a very similar task, even the abstract high-level features are probably very relevant, so you might tune just the classification head.

Transfer learning illustration (2)

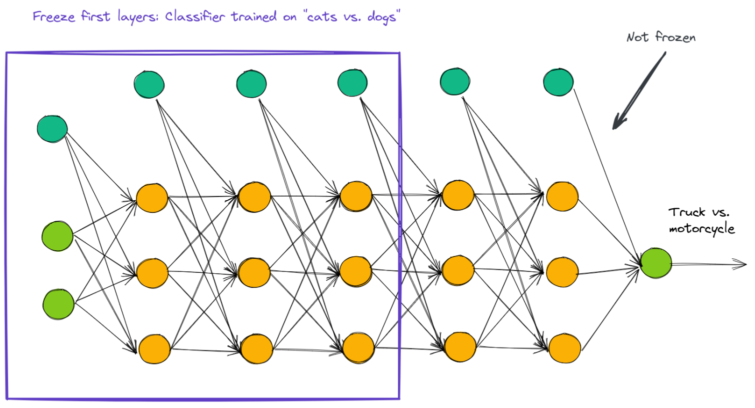

Figure 9: If the original network is not as relevant, may fine-tune more layers.

Architecture/setup

Recall: activation functions

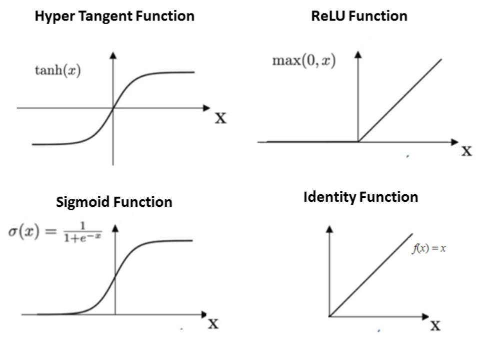

Figure 10: Candidate activation functions for a neural network.

Vanishing gradient

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.

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…

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

Same issue with tanh, although that is slightly better - its output is centered at zero, and its gradient has.

(There is also an analagous "exploding gradient" problem when large gradients are propagated back through the network.)

Dead ReLU

ReLU is a much better non-linear function:

- does not saturate in positive region
- very very fast to compute
- often converges faster than sigmoid/tanh

But, can "die" in the negative region.

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 in the negative region - won't die.

$$f(x) = \text{max}(\alpha x, x)$$

(α is a hyperparameter.)

Many other variations on this…

Weight initialization

What if we initialize weights to:

- zero?
- a constant (non-zero)?
- a normal random value with large σ?
- a normal random value with small σ?

Some comments:

- If weights are all initialized to zero, all the outputs are zero (for any input) - the network won't learn.
- 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".
- Large normal random values are bad - you want to be near the non-linear part of the activation function, and avoid exploding gradients.
- Small normal random values work well for "shallow" networks, but not for deep networks - it makes the activation function outputs "collapse" toward zero.

Desirable properties for initial weights

- The mean of the intial weights should be right in the middle
- The variance of the activations should stay the same across every layer (derivation)

Xavier initialization for tanh, He initialization for ReLU.

Xavier scales by $\frac{1}{\sqrt{N_{in}}}$, He by $\frac{2}{\sqrt{N_{in}}}$ where N_{in} is the number of inputs to the layer.

Desirable properties - illustration (1)

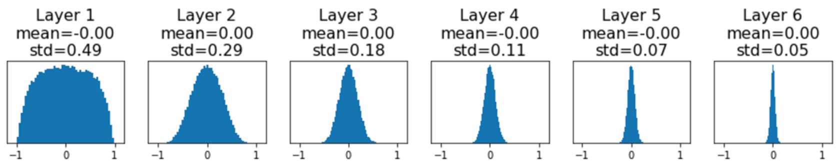

Figure 11: Activation function outputs with normal initialization of weights. Image source: Justin Johnson.

Desirable properties - illustration (2)

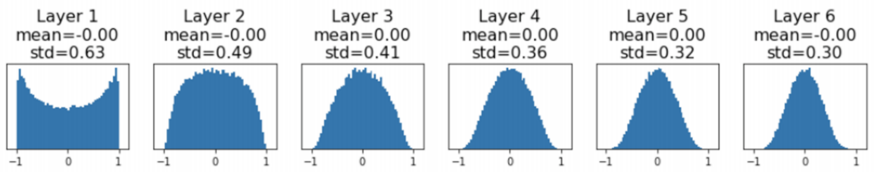

Figure 12: Activation function outputs with Xavier initialization of weights. Image source: Justin Johnson.

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.

Note: Whitening/decorrelation is not applied to image data. For image data, we sometimes subtract the "mean image" or the per-color mean.

Data preprocessing (1)

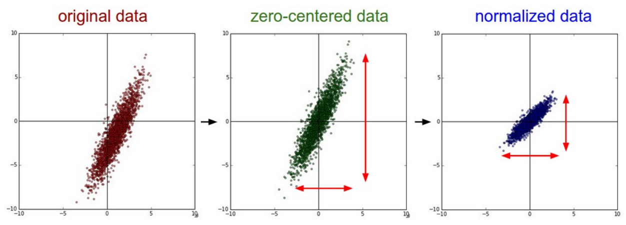

Figure 13: Image source: Stanford CS231n.

Data preprocessing (2)

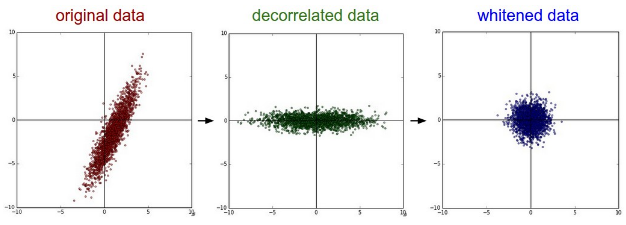

Figure 14: Image source: Stanford CS231n.

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

Batch normalization

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

Lots of discussion about how/why BatchNorm helps - still ongoing.

Gradient descent

Regularization

L1 or L2 regularization

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

- L1 penalty
- L2 penalty
- Combination (ElasticNet)

Not so common with neural networks.

Early stopping

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

Why does it work? Some ideas:

- The network is effectively "smaller" when we stop training early, because many units still in linear region of activation.
- Earlier layers (which learn simpler features) and late layers (near the output - used for response mapping) converge to their final weights first. See Boaz Barak.

Dropout

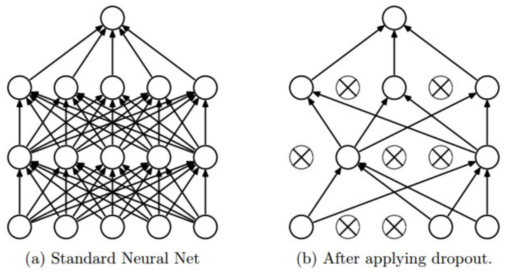

Figure 15: Dropout networks.

- 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)

Alternative: DropConnect zeros weights, instead of neurons.

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