Deep neural networks

Fraida Fund

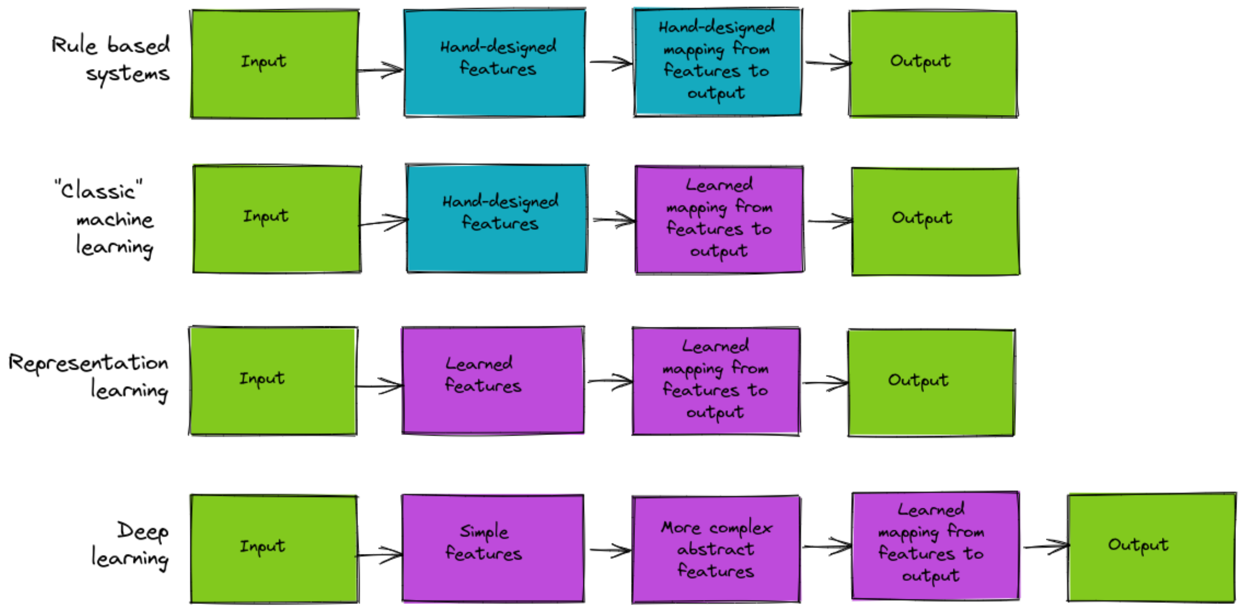
Contents

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.

Deep neural networks

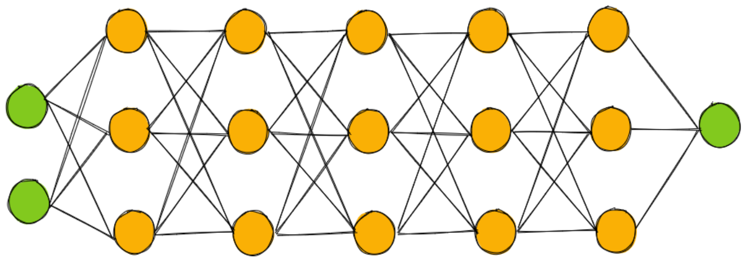


Figure 2: Illustration of a deep network, with multiple hidden layers.

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

- Optimization
- Generalization

Optimizing deep neural networks

Loss landscape

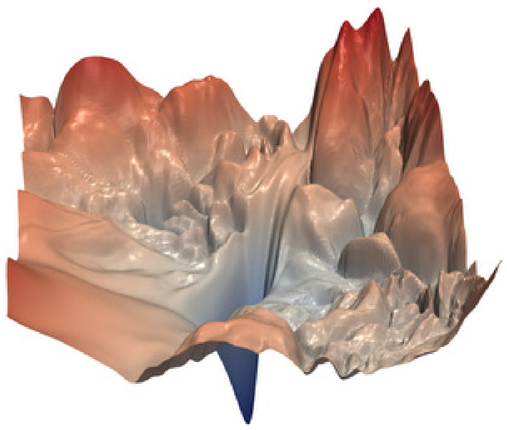


Figure 3: "Loss landscape" of a deep neural network in a "slice" of the high-dimensional feature space.

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!)

Effective training depends on

- Activation function
- Pre-processing data
- Initial weights
- Optimizer

Recall: activation functions

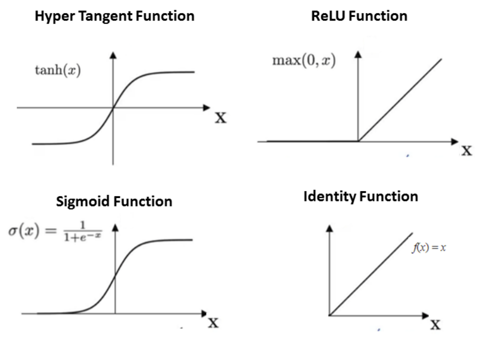


Figure 4: Candidate activation functions for a neural network.

Vanishing/exploding 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.

When the sigmoid “saturates”, it “kills” the neuron!

(Same issue with tanh.)

(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.)

Also other variations on this...

Skip connections

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

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.

What about zero-centering and normalization? Think about a binary classification problem of a data cloud that is far from the origin, vs. one close to the origin. In which case will the loss function react more (be more sensitive) to a small change in weights?

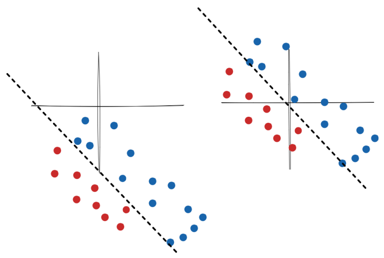


Figure 5: The classifier on the right is more sensitive to small changes in the weights.

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)

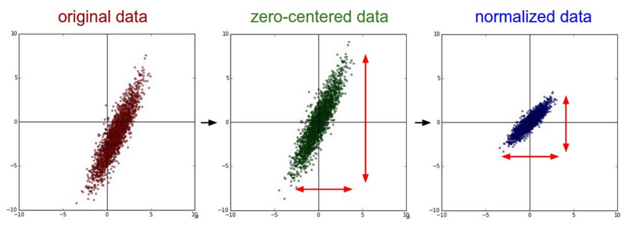


Figure 6: Image source: Stanford CS231n.

Data preprocessing (2)

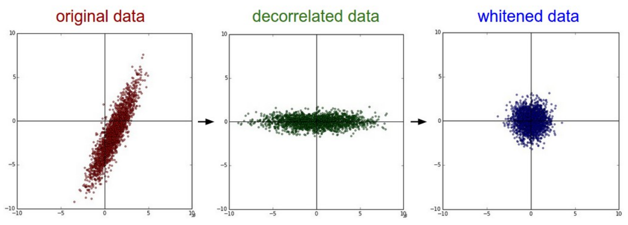


Figure 7: Image source: Stanford CS231n.

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

Desirable properties - illustration (1)

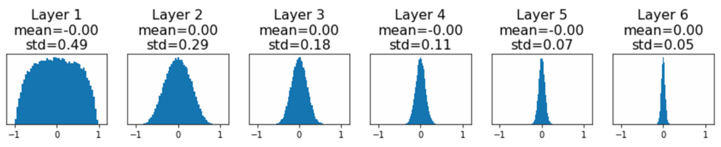


Figure 8: Activation function outputs with normal initialization of weights. Image source: Justin Johnson.

Desirable properties - illustration (2)

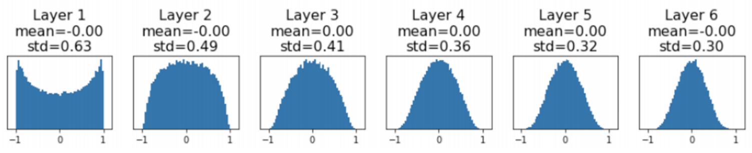


Figure 9: Activation function outputs with Xavier initialization of weights. Image source: Justin Johnson.

Optimizers

Standard ("batch") gradient descent

For each step t along the error curve:

$$W^{t+1} = W^t - \alpha \nabla L(W^t) = W^t - \frac{\alpha}{N} \sum_{i=1}^{N} \nabla L_i(W^t, \mathbf{x}_i, y_i)$$

Repeat until stopping criterion is met.

Stochastic gradient descent

Idea: at each step, compute estimate of gradient using only one randomly selected sample, and move in the direction it indicates.

Many of the steps will be in the wrong direction, but progress towards minimum occurs *on average*, as long as the steps are small.

Bonus: helps escape local minima.

Mini-batch (also "stochastic") gradient descent

Idea: In each step, select a small subset of training data ("mini-batch"), and evaluate gradient on that mini-batch. Then move in the direction it indicates.

For each step t along the error curve:

- Select random mini-batch $I_t \subset 1, \dots, N$
- Compute gradient approximation: $g^t = \frac{1}{|I_t|} \sum_{i \in I_t} \nabla L(\mathbf{x}_i, y_i, W)$
- Update parameters: $W^{t+1} = W^t - \alpha^t g^t$

Comparison: batch size

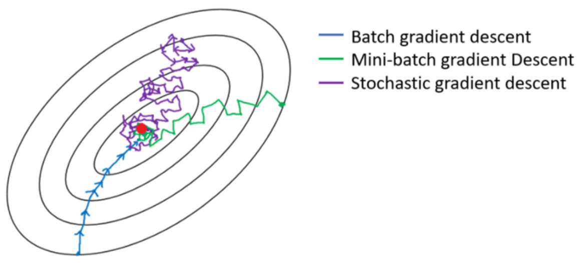


Figure 10: Effect of batch size on gradient descent.

Gradient descent terminology

- Mini-batch size is B, training size is N
- A training *epoch* is the sequence of updates over which we see all non-overlapping mini-batches
- There are $\frac{N}{B}$ steps per training epoch
- Data shuffling: at the beginning of each epoch, randomly shuffle training samples. Then, select mini-batches in order from shuffled samples.

Selecting the learning rate

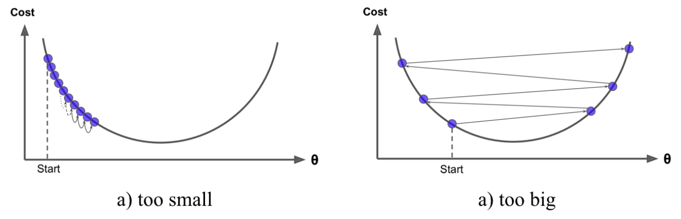


Figure 11: Choice of learning rate α is critical

Annealing the learning rate

One approach: decay learning rate slowly over time, such as

- Exponential decay: $\alpha_t = \alpha_0 e^{-kt}$
- 1/t decay: $\alpha_t = \alpha_0/(1 + kt)$

(where k is tuning parameter).

Gradient descent in a ravine (1)

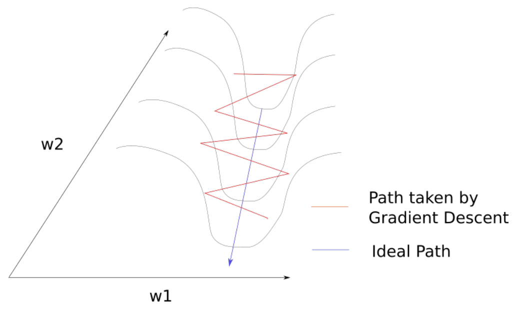


Figure 12: Gradient descent path bounces along ridges of ravine, because surface curves much more steeply in direction of w_1.

Gradient descent in a ravine (2)

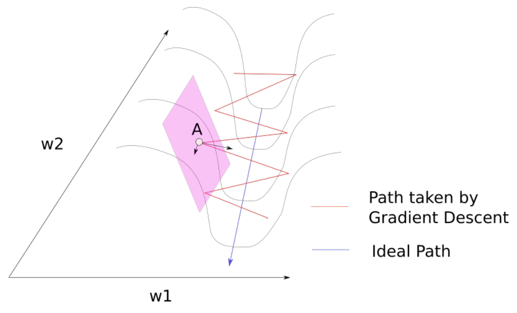


Figure 13: Gradient descent path bounces along ridges of ravine, because surface curves much more steeply in direction of w_1.

Momentum (1)

- Idea: Update includes a *velocity* vector v, that accumulates gradient of past steps.
- Each update is a linear combination of the gradient and the previous updates.
- (Go faster if gradient keeps pointing in the same direction!)

Momentum (2)

Classical momentum: for some $0 \leq \gamma_t < 1$,

$$v_{t+1} = \gamma_t v_t - \alpha_t \nabla L\left(W_t\right)$$

so

$$W_{t+1} = W_t + v_{t+1} = W_t - \alpha_t \nabla L\left(W_t\right) + \gamma_t v_t$$

(γ_t is often around 0.9, or starts at 0.5 and anneals to 0.99 over many epochs.)

Momentum: illustrated

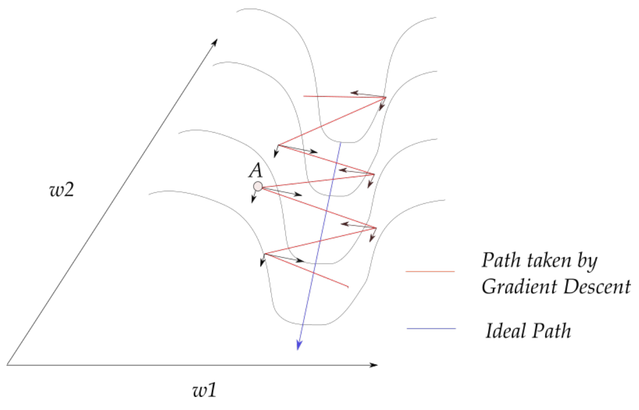


Figure 14: Momentum dampens oscillations by reinforcing the component along w_2 while canceling out the components along w_1.

RMSProp

Idea: Track *per-parameter* EWMA of *square* of gradient, and use it to adapt learning rate.

$$W_{t+1,i} = W_{t,i} - \frac{\alpha}{\sqrt{\epsilon + E[g^2]_t}} \nabla L(W_{t,i})$$

where

$$E[g^2]_t = (1-\gamma)g^2 + \gamma E[g^2]_{t-1}, \quad g = \nabla J(W_{t,i})$$

Weights with recent gradients of large magnitude have smaller learning rate, weights with small recent gradients have larger learning rates.

RMSProp: illustrated (Beale's function)

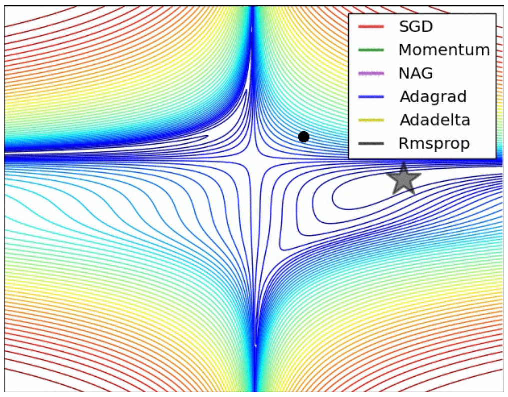


Figure 15: Animation credit: Alec Radford. Link to view animation.

Due to the large initial gradient, velocity based techniques shoot off and bounce around, RMSProps proceed more like faster SGD.

RMSProp: illustrated (Long valley)

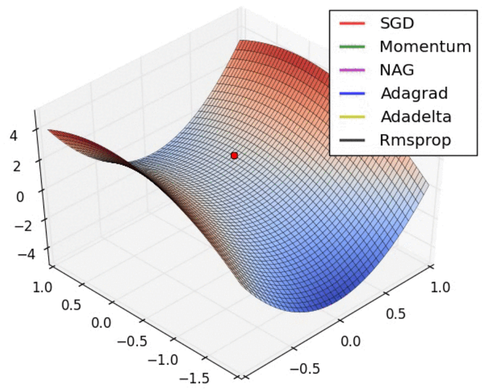


Figure 16: Animation credit: Alec Radford. Link to view animation.

SGD stalls and momentum has oscillations until it builds up velocity in optimization direction. Algorithms that scale step size quickly break symmetry and descend in optimization direction.

Adam: Adaptive moments estimation (2014)

Idea: Track the EWMA of *both* first and second moments of the gradient, $\{m_t, v_t\}$ at each time t.

If $L_t(W)$ is evaluation of loss function on a mini-batch of data at time t,

$$\begin{aligned} \{m_t, v_t\}, \mathbb{E}[m_t] &\approx \mathbb{E}[\,\nabla L_t(W)\,], \mathbb{E}[v_t] \\ &\approx \mathbb{E}[\,(\nabla L_t(W))^2\,] \end{aligned}$$

Scale α by $\frac{m_t}{\sqrt{v_t}}$ at each step.

Generalization

Why don’t deep neural networks overfit?

Double descent curve

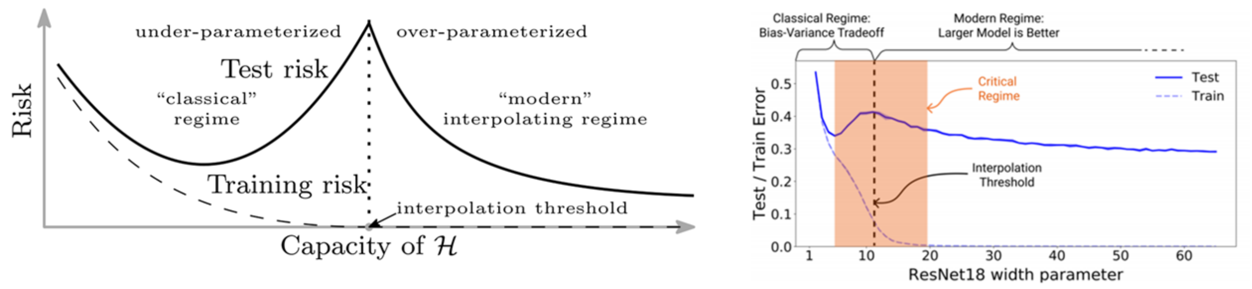


Figure 17: Double descent curve (left) and realization in a real neural network (right).

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

Double descent: animation

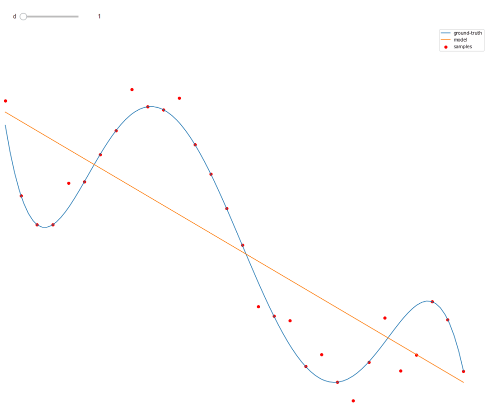


Figure 18: Polynomial model before and after the interpolation threshold. Image source: Boaz Barak, click link to see animation.

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.

Regularization

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

- L1 penalty
- L2 penalty
- Combination (ElasticNet)

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

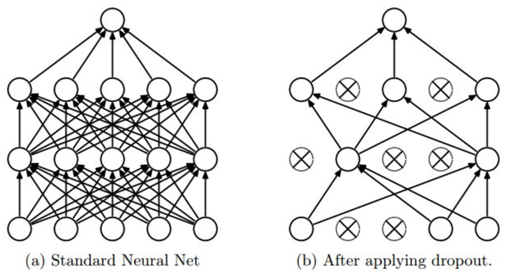


Figure 19: Dropout networks.

- 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?

Batch normalization

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

Data augmentation

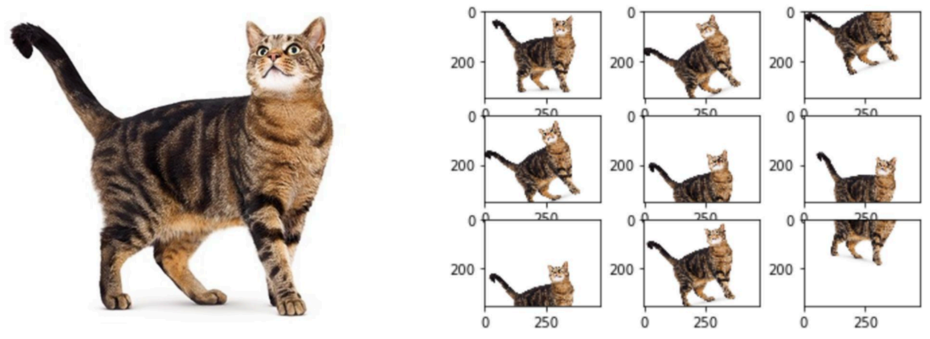


Figure 20: Data augmentation on a cat image.

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.