Lecture 19: Practical Concerns for Training Neural Networks

COMP 411, Fall 2021

Victoria Manfredi

WESLEYAN
UNIVERSITY

Acknowledgements: These slides are based primarily on slides created by Vivek Srikumar (Utah), Dan Roth (Penn), and Russ Grenier (U of Alberta) and content from the book “Machine Learning” by Tom Mitchell

Today's Topics

Training a neural network

- Initializing weights
- Data normalization
- Issues with stochastic gradient descent
- Preventing overfitting

vumanfredi@wesleyan.edu

Training

INITIALIZING WEIGHTS

Initializing weights

Initialize weights randomly, but close to zero

Give random number generator same random seed during debugging, to ensure you get the same output

Once debugging done: set random seed randomly, such as a function of current time

Training

DATA NORMALIZATION

Data normalization

Noise in large-valued features can be more than size of small-valued features!

Normalization

- Typically normalize so between -1 and 1 or 0 and 1
- May just normalize features by max-min
- Or may normalize based on distribution of features
  - e.g., many features in one range of values but few in another range

Training

LEARNING AS LOSS MINIMIZATION

Learning as loss minimization

The setup

- Examples x drawn from a fixed, unknown distribution D
- Hidden oracle classifier f labels examples
- We wish to find a hypothesis h that mimics f

Learning as loss minimization

The setup

- Examples x drawn from a fixed, unknown distribution D
- Hidden oracle classifier f labels examples
- We wish to find a hypothesis h that mimics f

The ideal situation

- Define a function L that penalizes bad hypotheses
- Learning: pick a function $h \in H$ to minimize expected loss

$$\min_{h \in H} E_{[x \sim D]}[L(h(x), f(x))]$$

But distribution D is unknown

Instead, minimize *empirical loss* on the training set

$$\min_{h \in H} \frac{1}{m} \sum_{i} L(h(x_i), f(x_i))]$$

Empirical loss minimization

Learning = minimize *empirical loss* on the training set

$$\min_{h \in H} \frac{1}{m} \sum_{i} L(h(x_i), f(x_i))]$$

Is there a problem here? Overfitting!

We need something that biases the learner towards simpler hypotheses

- Achieved using a regularizer, which penalizes complex hypotheses

Regularized loss minimization

Learning:

$$\min_{h \in H} \left(\text{regularizer}(h) + C \frac{1}{m} \sum_i L(h(x_i), f(x_i)) \right)$$

With linear classifiers:

$$\min_{\mathbf{w}} \mathbf{w}^T \mathbf{w} + \frac{C}{m} \sum_i L(y_i, \mathbf{x}_i, \mathbf{w})$$

What is a loss function?

- Loss functions should penalize mistakes
- We are minimizing average loss over the training data

What is the ideal loss function for classification?

The 0-1 loss

Penalize classification mistakes between true label y and prediction y'

$$L_{0-1}(y, y') = \begin{cases} 1 & \text{if } y \neq y' \\ 0 & \text{if } y = y' \end{cases}$$

For linear classifiers, the prediction $y' = \text{sgn}(\mathbf{w}^T\mathbf{x})$

- Mistake if $y\mathbf{w}^T\mathbf{x} \leq 0$

$$L_{0-1}(y, y') = \begin{cases} 1 & \text{if } y\mathbf{w}^T\mathbf{x} \leq 0 \\ 0 & \text{otherwise} \end{cases}$$

Minimizing 0-1 loss is intractable. Need surrogates

Learning via loss minimization

- Write down a loss function, minimize empirical loss
- Regularize to avoid overfitting
  - Neural networks use other strategies such as dropout
- Widely applicable, different loss functions and regularizers

Training

ISSUES WITH STOCHASTIC GRADIENT DESCENT

Comments on Training

No guarantee of convergence; may oscillate or reach a local minima.

In practice, many large networks are trained on large amounts of data for realistic problems.

Many epochs (tens of thousands) may be needed for adequate training. Large data sets may require many hours/days/weeks of CPU or GPU time, sometimes specialized hardware even

Termination criteria: Number of epochs; Threshold on training set error; No decrease in error; Increased error on a validation set.

To avoid local minima: several trials with different random initial weights with majority or voting techniques

Minibatches

Stochastic gradient descent

- Take a random example at each step
- Write down the loss function with that example
- Compute gradient of this loss and take a step

Why should we take only one random example at each step?

Stochastic gradient descent with minibatches

- Collect a small number of random examples (the minibatch) at each step
- Write down the loss function with that example
- Compute gradient of this loss and take a step

New hyperparameter: size of the mini batch

- Often governs how fast learning converges
- Hardware considerations around memory can dictate size of minibatch

Gradient tricks

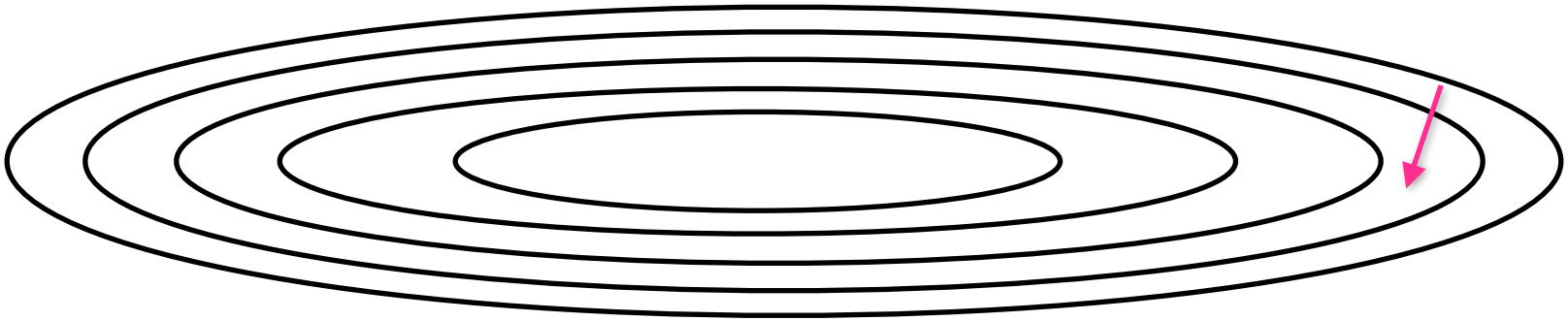
Simple gradient descent updates the parameters using the gradient of one example (or a mini batch of them), denoted by g_i

$$\text{parameters} \leftarrow \text{parameters} - \eta g_i$$

Gradients could change much faster in one direction than another

When gradients change very fast, this can make learning slow, or worse, unstable.

Quality of model can change drastically based on how many epoch you run



Each pink link line is gradient

Gradient tricks

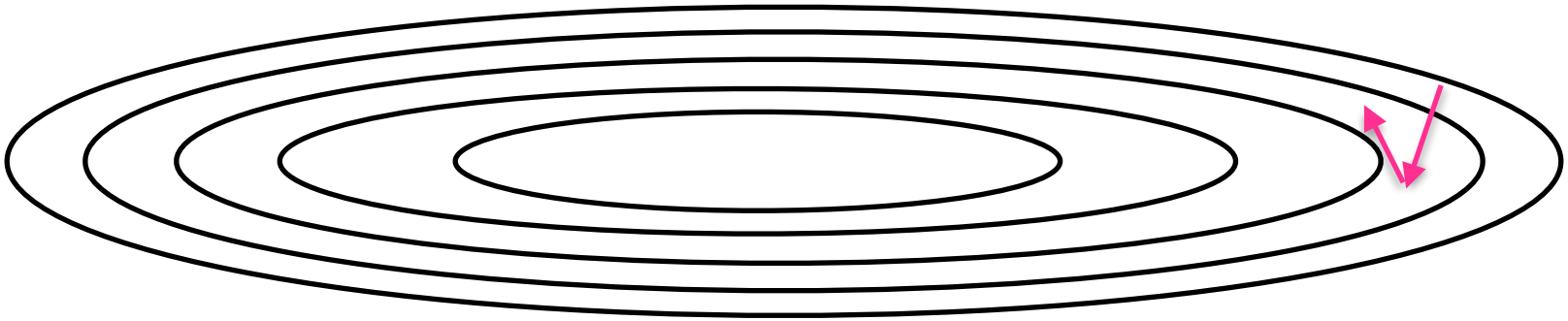
Simple gradient descent updates the parameters using the gradient of one example (or a mini batch of them), denoted by g_i

$$\text{parameters} \leftarrow \text{parameters} - \eta g_i$$

Gradients could change much faster in one direction than another

When gradients change very fast, this can make learning slow, or worse, unstable.

Quality of model can change drastically based on how many epoch you run



Each pink link line is gradient

Gradient tricks

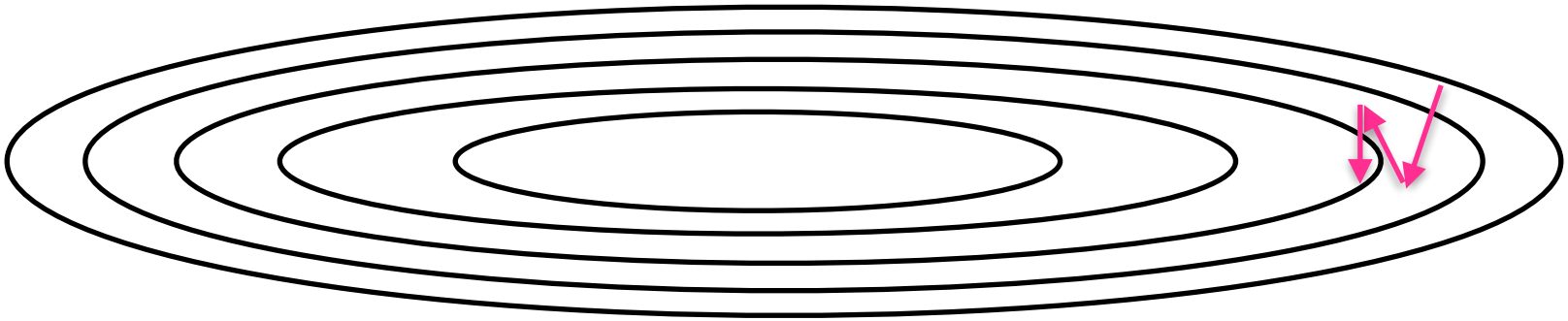
Simple gradient descent updates the parameters using the gradient of one example (or a mini batch of them), denoted by g_i

$$\text{parameters} \leftarrow \text{parameters} - \eta g_i$$

Gradients could change much faster in one direction than another

When gradients change very fast, this can make learning slow, or worse, unstable.

Quality of model can change drastically based on how many epoch you run



Each pink link line is gradient

Gradient tricks

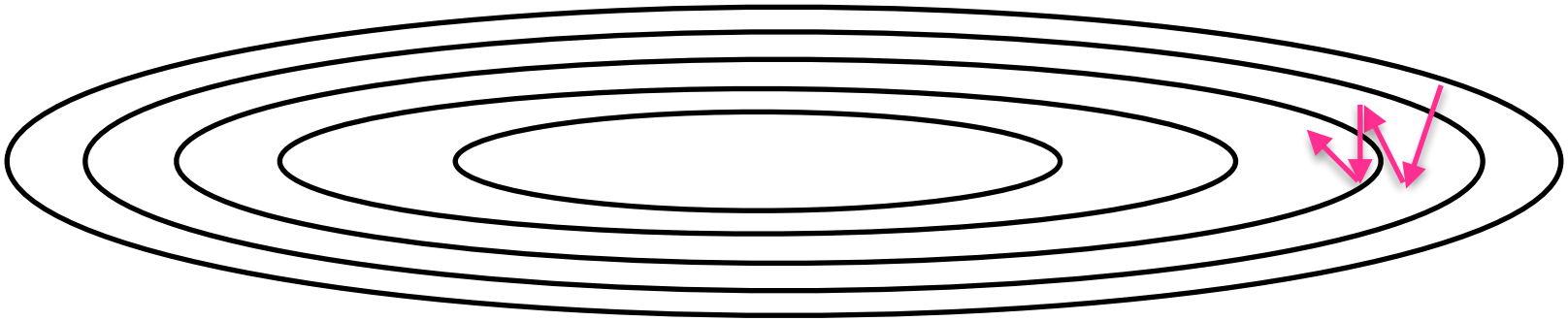
Simple gradient descent updates the parameters using the gradient of one example (or a mini batch of them), denoted by g_i

$$\text{parameters} \leftarrow \text{parameters} - \eta g_i$$

Gradients could change much faster in one direction than another

When gradients change very fast, this can make learning slow, or worse, unstable.

Quality of model can change drastically based on how many epoch you run



Each pink link line is gradient

Gradient tricks: momentum

Momentum smooths out updates by using a weighted average of all previous gradients at each step

Instead of updating with the gradient (g_i), use a moving average of gradients ($\mathbf{v}_t$) to update the model parameters. In the inner loop:

$$\mathbf{v}_t \leftarrow \mu \mathbf{v}_t - 1 + \eta_t g_i$$

$$\text{parameters} \leftarrow \text{parameters} - \mathbf{v}_t$$

Update is average of previous update and gradient

The hyperparameter μ controls how much of the previous update should be retained. Typical value $\mu = 0.9$

Gradient tricks: AdaGrad, RMSProp, Adam

AdaGrad. Each parameter has its own learning rate. If $g_{i,t}^2$ is the gradient for the ith parameter at step t, then

$$c_i \leftarrow c_i + g_{i,t}^2$$

$$\text{parameters}_i \leftarrow \text{parameters}_i - \frac{\eta}{\alpha + \sqrt{c_i}} g_{i,t}$$

RMSProp. Similar to AdaGrad but more recent gradients are weighted more in the denominator

$$c_i \leftarrow \delta c_i + (1 - \delta) g_{i,t}^2$$

Adam. A combination of many ideas:

- Momentum to smooth gradients
- RMSProp like approach for adaptively choosing learning rate with more recent gradients being weighted higher
- Additional terms to avoid bias introduced during early gradient estimates
- Currently the most commonly used variant of gradient based learning

Training a Neural Network

OVERFITTING PREVENTION

Over-fitting prevention: validation set

Running too many epochs may over-train the network and result in over-fitting (improved result on training, decrease in performance on test set)

Keep an hold-out validation set and test accuracy after every epoch

Maintain weights for best performing network on the validation set and return it when performance decreases significantly beyond that.

To avoid losing training data to validation:

- Use 10-fold cross-validation to determine the average number of epochs that optimizes validation performance
- Train on full data set using this many epochs to produce final results

Over-fitting prevention: hidden units

Too few hidden units prevent the system from adequately fitting the data and learning the concept

Using too many hidden units leads to over-fitting

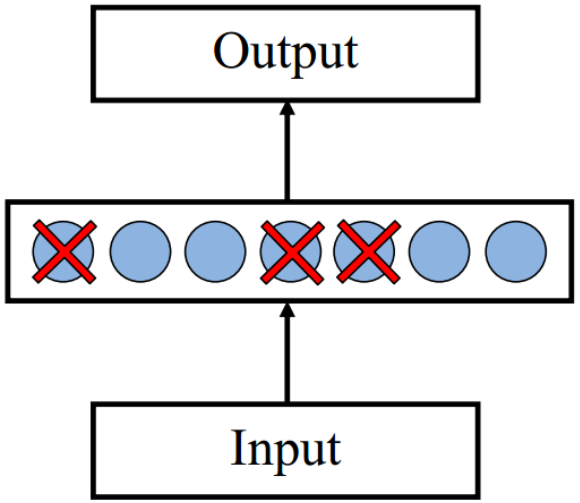
Cross-validation or performance on a held out set can be used to determine an appropriate number of hidden units

Over-fitting prevention: dropout training

Proposed by (Hinton et al, 2012)

During training, for each step, decide whether to delete a hidden unit with some probability p

- That is, make predictions using only a randomly chosen set of neurons
- Update only these neurons

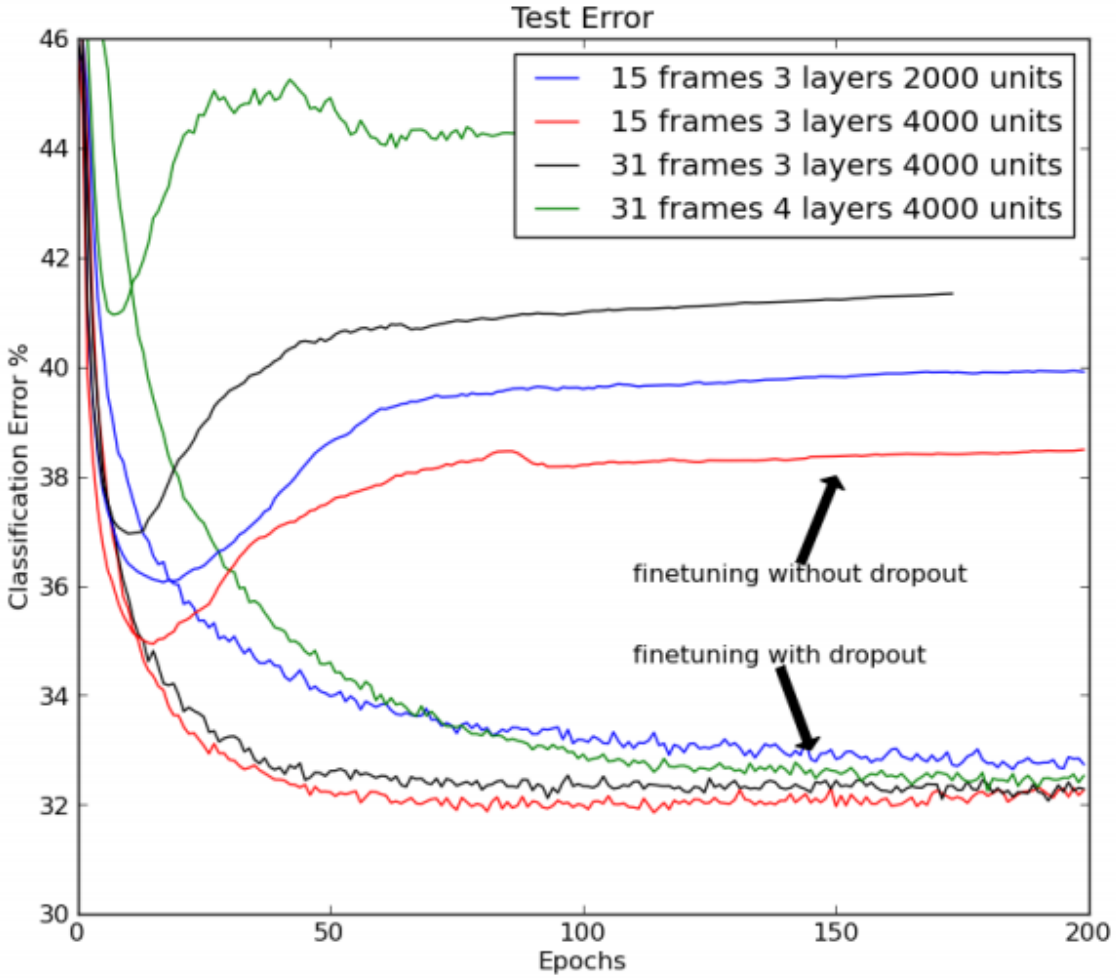


Tends to avoid overfitting

Has a model averaging effect

- Only some parameters get trained at any step

Over-fitting prevention: dropout training



Dropout of 50% of the hidden units and 20% of the input units (Hinton et al, 2012)

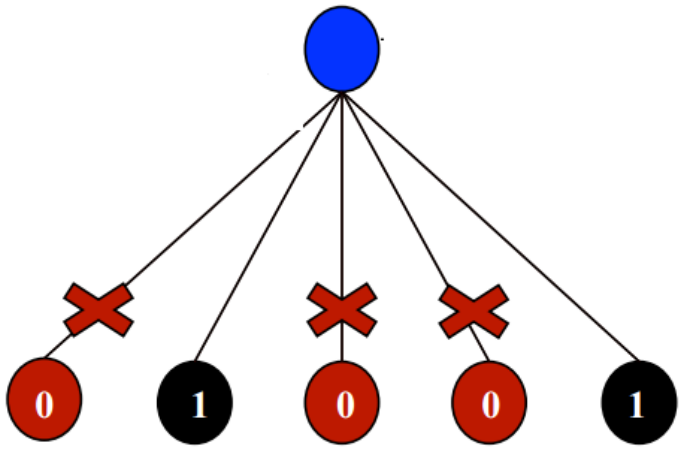
Over-fitting prevention: dropout training

Model averaging effect

- Among 2^H models, with shared parameters
  - H: number of units in the network
- Only a few get trained
- Much stronger than the known regularizer

What about the input space?

- Do the same thing!



Over-fitting prevention: weight-decay

All weights are multiplied by some fraction in (0,1) after every epoch

- Encourages smaller weights and less complex hypothesis
- Equivalently: change Error function to include a term for the sum of squares of weights in network

Training a Neural Network

PRACTICAL TIPS

Input-Output Coding

- Appropriate coding of inputs and outputs can make learning problem easier and improve generalization
- Encode each binary feature as a separate input unit
- For multi-valued features include one binary unit per value rather than trying to encode input information in fewer units
  - Very common today to use distributed representation of the input – real valued, dense representation
- For disjoint categorization problem, best to have one output unit for each category rather than encoding N categories into $\log N$ bits

Representational Power

Backpropagation version presented is for networks with one hidden layer

But:

- Any Boolean function can be represented by a two layer network (simulate a two layer AND-OR network)
- Any bounded continuous function can be approximated with arbitrary small error by a two layer network
- Sigmoid functions provide a set of basis function from which arbitrary function can be composed
- Any function can be approximated to arbitrary accuracy by a three layer network

Hidden Layer Representation

- Weight tuning procedure sets weights that define whatever hidden units representation is most effective at minimizing the error

- Sometimes Backpropagation will define new hidden layer features that are not explicit in the input representation, but which capture properties of the input instances that are most relevant to learning the target function

- Trained hidden units can be seen as newly constructed features that re-represent the examples so that they are linearly separable

Gradient checks are useful!

Allow you to know that there are no bugs in your neural network implementation!

- Implement your gradient

- Implement a finite difference computation by looping through the parameters of your network, adding and subtracting a small epsilon ($\sim 10^{-4}$) and estimate derivatives

$$f'(\theta) \approx \frac{f\left(\theta^{+}\right) - f(\theta^{-})}{2\epsilon} \qquad \theta^{\pm} = \theta \pm \epsilon$$

- Compare the two and make sure they are almost the same

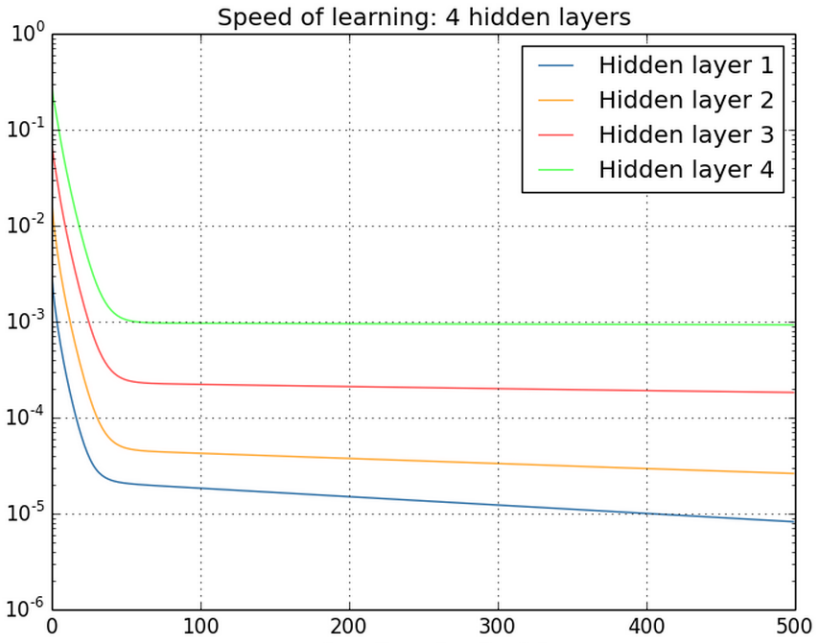
Vanishing/exploding gradients

Gradient can become very small or very large quickly, and the locality assumption of gradient descent breaks down (Vanishing gradient) [Bengio et al 1994]

Vanishing gradients are quite prevalent and a serious issue

A real example

- Training a feed-forward network
- y-axis: sum of the gradient norms
- Earlier layers have exponentially smaller sum of gradient norms
- This will make training earlier layers much slower



Why backpropagation tends to work?

Only guaranteed to converge

- eventually
- to a local optimum

Why does it work so well in practice?

- At start $w/w_{ij} \approx 0$, network $\approx$ linear weights, so moves quickly… until in "correct region"

Efficiency

Number of iterations: very important!

- If too small: higher error
- if too large: overfitting $\Longrightarrow$ high general error

Learning: intractable in general

- Training can take thousands of iterations: slow!
- Learning net with single hidden unit is NP-hard
- In practice: backpropagation is very useful

Use: Using network (after training) is very fast

Why deep neural networks?

MOTIVATION

Why deep neural networks?

Universality

- In principle can approximate an arbitrary function using just a single hidden layer.

Why should we use neural networks with many layers?

- Well-adapted to learning hierarchies of knowledge:

  pixel → shape → object → multiple objects → scene

Neural Networks

SUMMARY

What we saw

What is a neural network

Multiple layers:

- inner layers learn a representation of the data

Highly expressive

- Neural networks can learn arbitrarily complex functions
- Is this always a good thing? Overfitting?
- Can be challenging to learn the parameters as multiple optima. Many tricks to make gradient descent work

Training neural networks

- Backpropagation

What we did not see

Vast area, fast moving

- Many new algorithms and tricks for learning that tweak on the basic gradient method

Some named neural networks

- **Restricted Boltzmann machines and auto encoders**: learn a latent representation of the data
- **Convolutional neural network**: modeled after the mammalian visual cortex, currently the state of the art for object recognition tasks
- **Recurrent neural networks and transformers**: encode and predict sequences: we will look at in a few weeks!
- **Attention**: use a neural network to decide what parts of a set of features are relevant and create an aggregate "attended" representation
- … and many more