Lecture 20: Neural Networks Practical

COMP 343, Spring 2022

Victoria Manfredi

WESLEYAN
UNIVERSITY

Acknowledgements: These slides are based primarily on content from the book “Machine Learning” by Tom Mitchell and slides created by Vivek Srikumar (Utah) and Dan Roth (Penn) and Dan Klein and Pieter Abbeel (UC Berkeley)

Today's Topics

Homework 7

- due Wednesday, April 20

Recap

Neural networks practical concerns

- General tips
- Problems with gradient descent
- Preventing overfitting
- Summary

vumanfredi@wesleyan.edu

Recap

Backpropagation algorithm

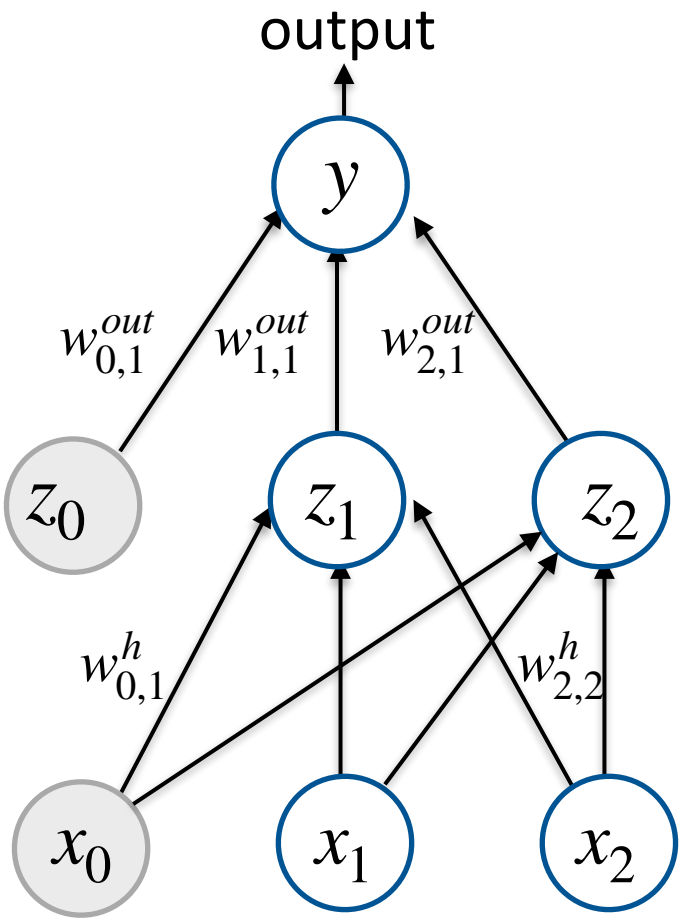
The same algorithm works for multiple layers and more complicated architectures

Repeated application of the chain rule for partial derivatives

- First perform forward pass from inputs to the output
- Compute loss
- From loss, proceed backwards to compute partial derivatives using chain rule
- Cache partial derivatives as you compute them to use for lower layers

Backpropagation (of errors)

$$L = \frac{1}{2}(y - y^*)^2$$



$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$

$$z_2 = \sigma(w^h_{0,2} + w^h_{1,2} x_1 + w^h_{2,2} x_2)$$

$$z_1 = \sigma(w^h_{0,1} + w^h_{1,1} x_1 + w^h_{2,1} x_2)$$

We want to compute $\frac{\partial L}{\partial w^o_{ij}}$ and $\frac{\partial L}{\partial w^h_{ij}}$

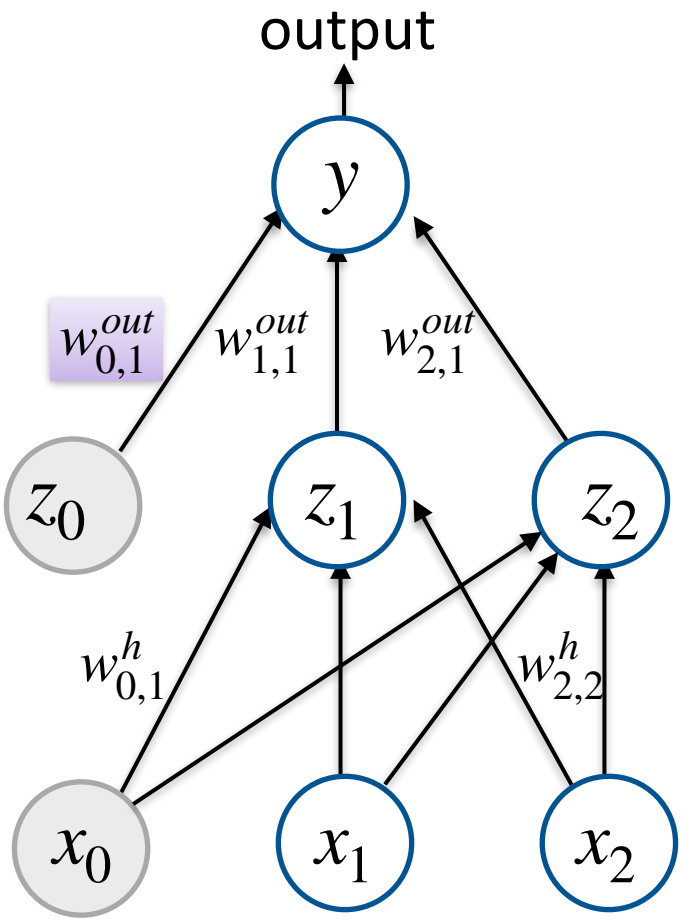
Important: L is a differentiable function of all of the weights

Applying the chain rule to compete the gradient (and remembering partial computations along the way to speed up learning)

Backpropagation: output layer

$$L = \frac{1}{2}(y - y^*)^2$$

$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$



$$\frac{\partial L}{\partial w^o_{0,1}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w^o_{0,1}}$$

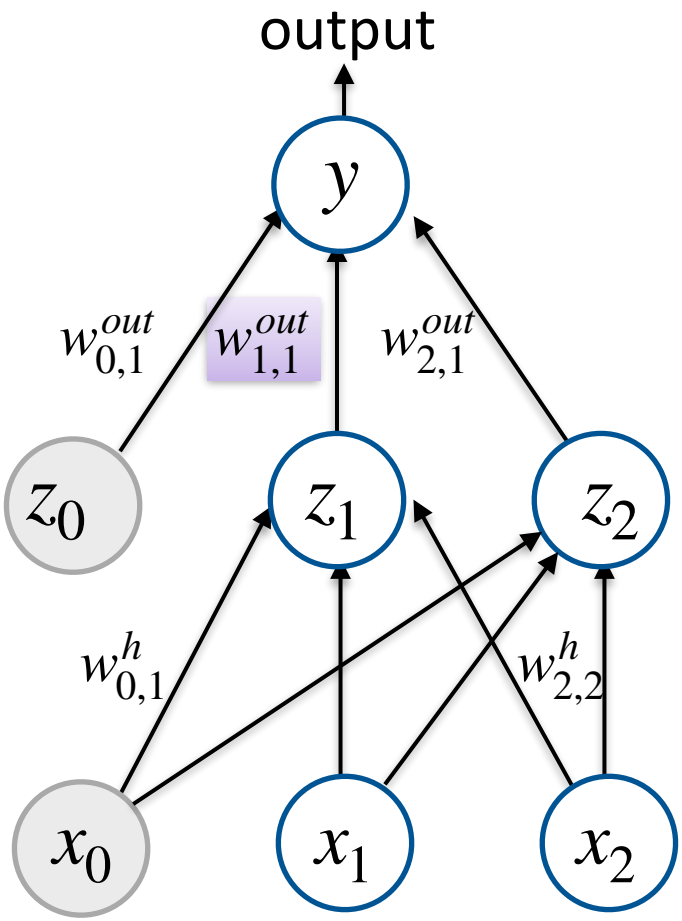
$$\frac{\partial L}{\partial y} = y - y^*$$

$$\frac{\partial y}{\partial w^o_{0,1}} = 1$$

Backpropagation: output layer

$$L = \frac{1}{2}(y - y^*)^2$$

$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$



$$\frac{\partial L}{\partial w^o_{1,1}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w^o_{1,1}}$$

$$\frac{\partial L}{\partial y} = y - y^*$$

$$\frac{\partial y}{\partial w^o_{1,1}} = z_1$$

We have already computed this partial derivative for the previous case. Cache to speed up!

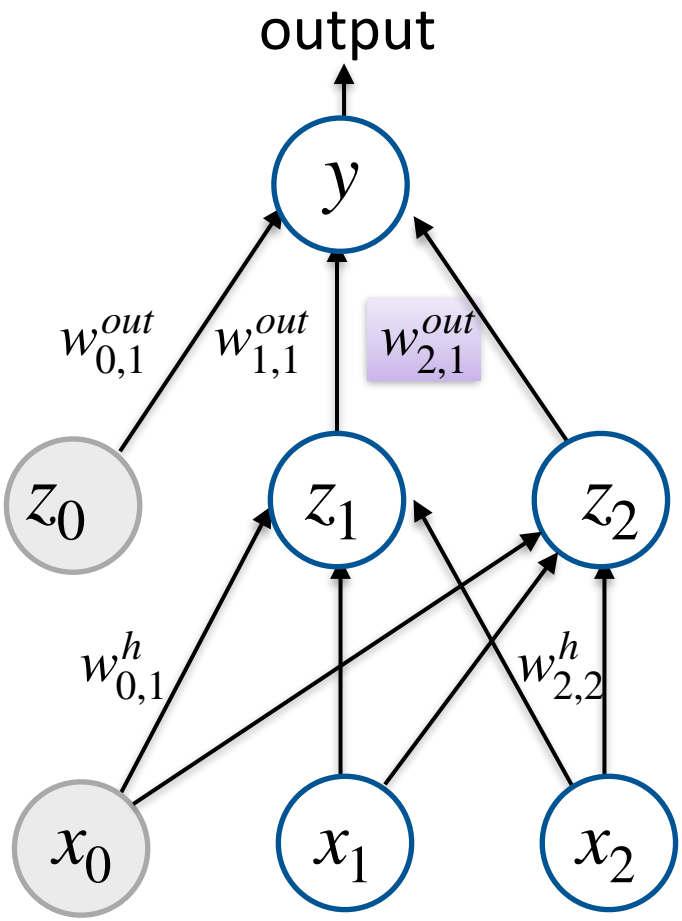
Where do we get value of z_1 from?

Compute during forward pass!
If you compute something keep it around, may be useful

3 min: what is $\partial L / \partial w^o_{2,1}$?

$$L = \frac{1}{2}(y - y^*)^2$$

$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$



$$\frac{\partial L}{\partial w^o_{2,1}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w^o_{2,1}}$$

$$\frac{\partial L}{\partial y} = y - y^*$$

$$\frac{\partial y}{\partial w^o_{2,1}} = z_2$$

Computed during forward pass!

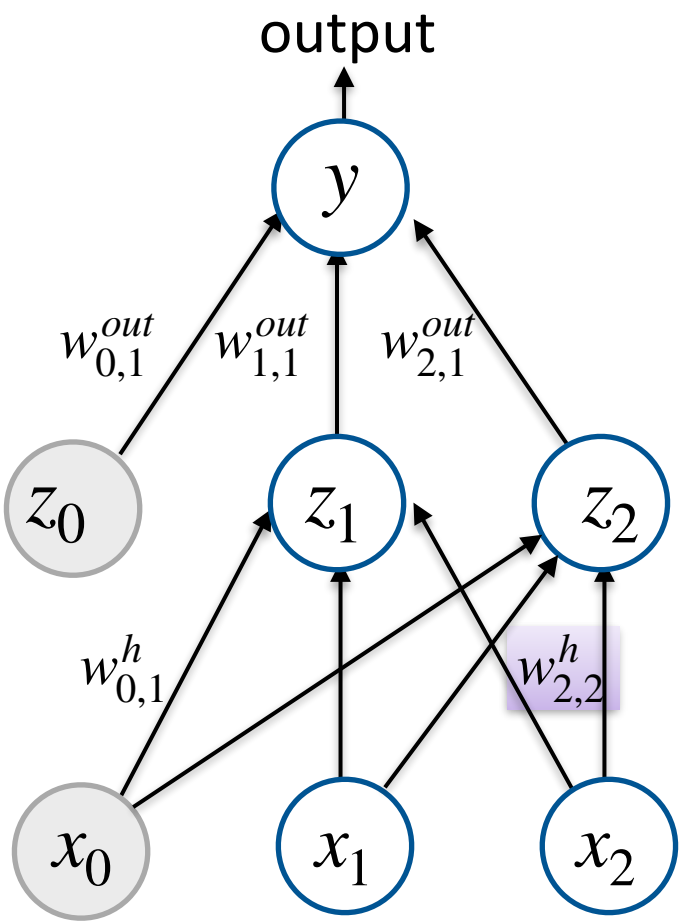
Backpropagation: hidden layer

$$L = \frac{1}{2}(y - y^*)^2$$

$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$

$$z_2 = \sigma(w^h_{0,2} + w^h_{1,2} x_1 + w^h_{2,2} x_2)$$

$$z_1 = \sigma(w^h_{0,1} + w^h_{1,1} x_1 + w^h_{2,1} x_2)$$



$$\frac{\partial L}{\partial w^h_{2,2}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w^h_{2,2}}$$

So apply chain rule

$$= \frac{\partial L}{\partial y} \cdot \frac{\partial}{\partial w^h_{2,2}}(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$

Substitute in for y

$$= \frac{\partial L}{\partial y} \cdot (w^o_{1,1} \frac{\partial}{\partial w^h_{2,2}} z_1 + w^o_{2,1} \frac{\partial}{\partial w^h_{2,2}} z_2)$$

Derivative of sum is sum of derivatives

$\frac{\partial}{\partial w^h_{2,2}} z_1 \rightarrow 0$

z_1 is not a function of $w^h_{2,2}$ so can eliminate term (like a constant)

Backpropagation: hidden layer

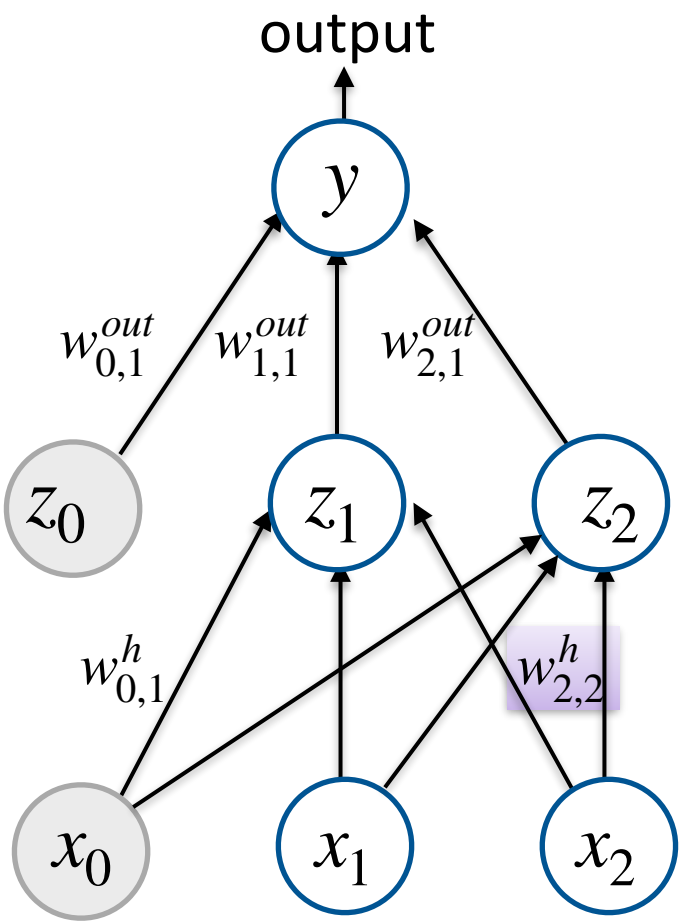
$$L = \frac{1}{2}(y - y^*)^2$$

$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$

$$z_2 = \sigma(w^h_{0,2} + w^h_{1,2} x_1 + w^h_{2,2} x_2)$$

$$z_1 = \sigma(w^h_{0,1} + w^h_{1,1} x_1 + w^h_{2,1} x_2)$$

$\sigma(s)$



from previous slide

$$\frac{\partial L}{\partial w^h_{2,2}} = \frac{\partial L}{\partial y} \cdot w^o_{2,1} \frac{\partial z_2}{\partial s} \frac{\partial s}{\partial w^h_{2,2}}$$

Compute gradient of z_2 with respect to s

Each of these partial derivatives is easy!

Backpropagation: hidden layer

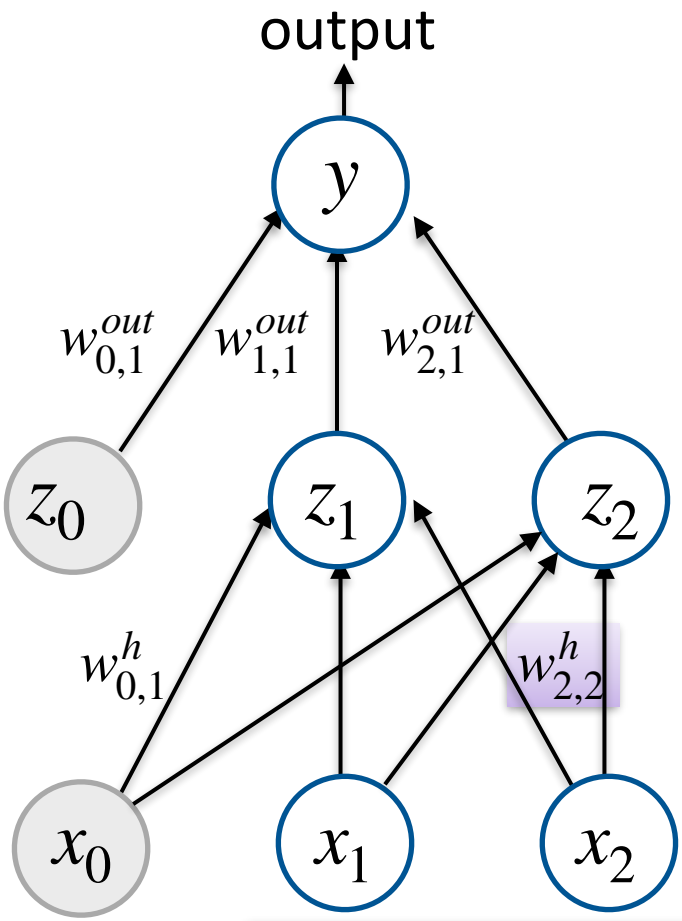
$$L = \frac{1}{2}(y - y^*)^2$$

$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$

$$z_2 = \sigma(w^h_{0,2} + w^h_{1,2} x_1 + w^h_{2,2} x_2)$$

$$z_1 = \sigma(w^h_{0,1} + w^h_{1,1} x_1 + w^h_{2,1} x_2)$$

$\sigma(s)$



from previous slide

$$\frac{\partial L}{\partial w^h_{2,2}} = \frac{\partial L}{\partial y} \cdot w^o_{2,1} \frac{\partial z_2}{\partial s} \frac{\partial s}{\partial w^h_{2,2}}$$

$$\frac{\partial L}{\partial y} = y - y^*$$

$$\frac{\partial z_2}{\partial s} = z_2(1 - z_2)$$

$$\frac{\partial s}{\partial w^h_{2,2}} = x_2$$

Because $z_2 = \sigma(s) = \frac{1}{1 + e^{-s}}$ is the sigmoid function which has the derivative $\sigma(s)(1 - \sigma(s))$

Importantly: we have already computed many of these partial derivatives because we are proceeding from top to bottom (i.e., backwards). And calculations can be vectorized for efficient computation on GPUs

Stochastic gradient descent

$$\min_{w} \sum_{i} L(NN(\mathbf{x_i}, \mathbf{w}), y_i)$$

The objective is not convex
Initialization can be important

Given a training set $S = \{(\mathbf{x_i}, y_i)\}, \mathbf{x} \in \Re^{\mathbf{d}}$

1. Initialize parameters $\mathbf{w}$
2. For epoch $= 1 \ldots T$:
   - Shuffle the training set
   - For each training example $(\mathbf{x_i}, y_i) \in S$:
     - ➡ Treat this example as the entire dataset

       Compute the gradient of the loss $\nabla L(NN(\mathbf{x_i}, \mathbf{w}), y_i)$ using backpropagation

     - ➡ Update $\mathbf{w} \leftarrow \mathbf{w} - \gamma_t \nabla L(NN(\mathbf{x}_i, \mathbf{w}), y_i)$

γ_t: learning rate, many tweaks possible

Return $\mathbf{w}$

Neural Networks

OTHER LOSS FUNCTIONS (AND ACTIVATIONS)

Regression vs. classification

3 min: What should we change with neural network?

Regression

- Output activation: Linear, Rectified Linear (ReLU), …
- Squared loss

Classification

- Output activation: sigmoid, …
- What loss function to use? One option is Log loss

Important: changing loss function or activations will change gradient computations.

Log loss

Sigmoid activation on output (i.e., $y_{pred} = \sigma(z(\mathbf{x}))$) makes log loss easy to compute as output can be interpreted as probability of label.

- $\sigma(z(\mathbf{x}))$: probability $\mathbf{x}$ belongs to positive class
- $1 - \sigma(z(\mathbf{x}))$: probability $\mathbf{x}$ belongs to negative (or other) class

Thus log loss for binary classification with sigmoid activation is

$$\text{Log loss} = -\left(y_i^{true}\log(y_i^{pred}) + (1 - y_i^{true})\log(1 - y_i^{pred})\right)$$

In homework 7 you will compute log loss averaged over all examples (this in preparation for implementing Backpropagation on homework 8)

$$\text{Log loss} = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i^{true}\log(y_i^{pred}) + (1 - y_i^{true})\log(1 - y_i^{pred})\right)$$

true probability of label

predicted probability of label

Neural Networks

GENERAL TIPS

How to initialize 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

Question: why is randomness particularly important now?

How to normalize data?

Noise in large-valued features can be more than size of small-valued features! So want at a minimum all features to have values within same range

Normalization

- Typically normalize 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

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

Neural Networks

PROBLEMS WITH GRADIENT DESCENT

How to handle lack of convergence?

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

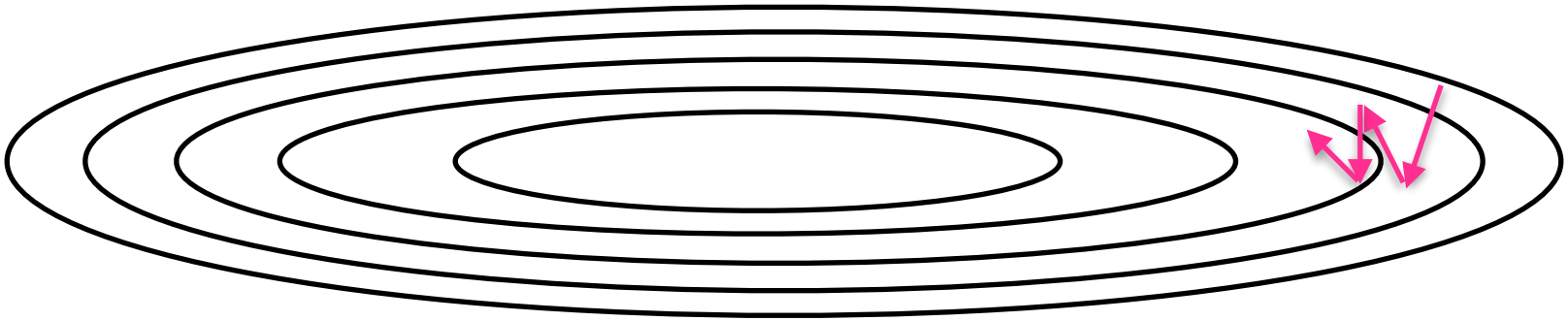
Simple gradient descent updates the parameters using the gradient of one example (or a mini batch of them), denoted by g_i

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

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

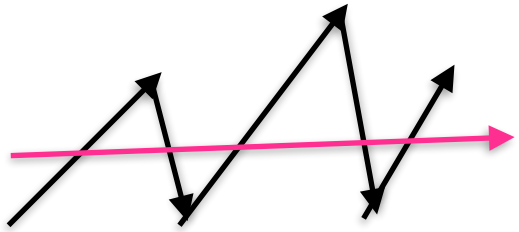


Each pink link line is gradient

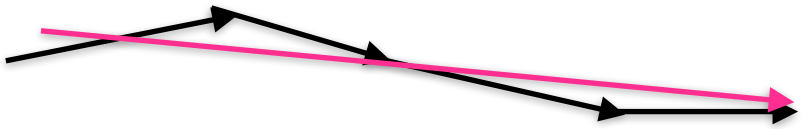
Gradient tricks: momentum

Averaging together successive gradients seem to yield a much better direction

Intuition: if successive gradients steps point in different directions we should cancel off the directions that disagree



If successive gradient steps point in similar directions, we should go faster in that direction



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:

$$\mathbf{v}_t \leftarrow \mu \mathbf{v}_{t-1} + (1-\mu)\eta_t g_i$$

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

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

- While direction of gradient is useful, magnitude may be very variable from one update to the next.

- So normalizing by a running average of gradients seen so far can be useful, so that every update is approximately the same size, rather than sometimes big updates and small updates

- AdaGrad, RMSProp, Adam are versions of gradient descent that incorporate these ideas

Gradient tricks: AdaGrad

AdaGrad. Estimate per-dimension cumulative magnitude, then divide each dimension by its magnitude. Each parameter (weight) has its own learning rate

RMSProp. Estimate per-dimension magnitude (running average), then divide each dimension by its magnitude. Similar to AdaGrad but more recent gradients are weighted more in the denominator

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

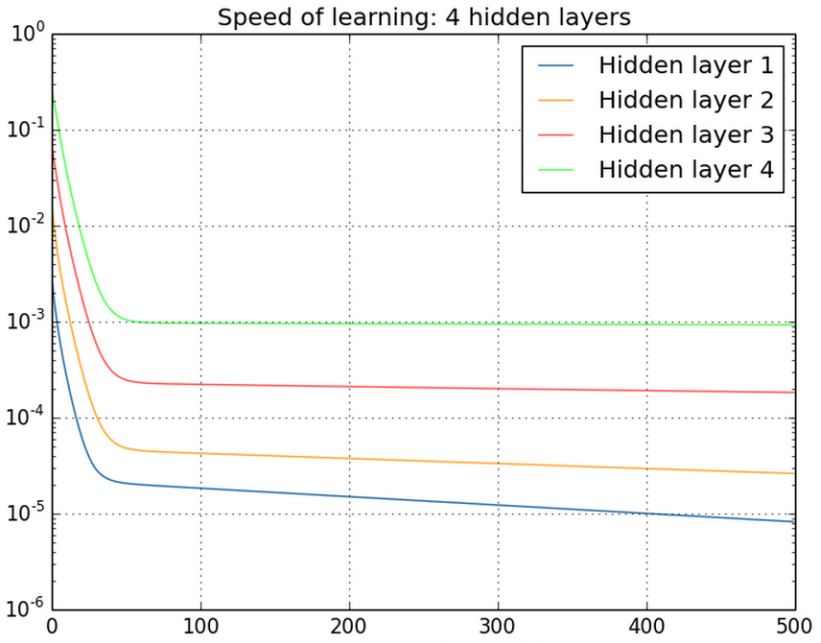
Vanishing/exploding gradients

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



Try changing activation functions, architectures, learning weights, weight initialization …

Neural Networks

PREVENTING OVERFITTING

Preventing overfitting: why?

As you make neural networks deeper (many hidden layers), adding complexity to model. Maybe enough complexity to just memorize data.

What do we know about very complex models?

Prone to overfitting!

Preventing overfitting: how?

1. Cross-validation

2. Changing number of hidden units

3. Dropout training

Preventing overfitting: 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

Preventing overfitting: 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

Preventing overfitting: dropout training

Proposed by (Hinton et al, 2012)

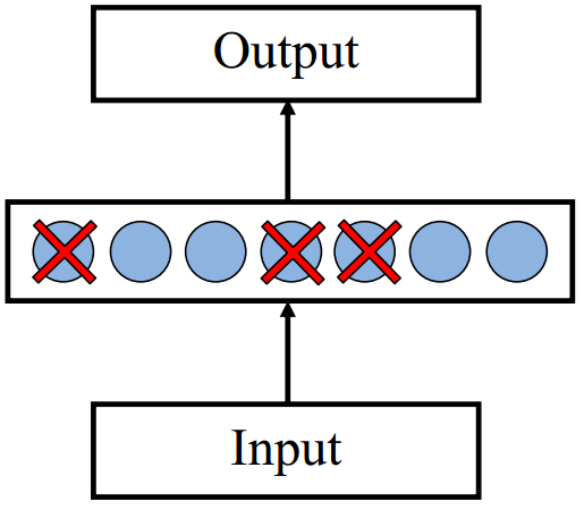
During training, for each step, decide whether to delete a hidden unit with some probability p

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

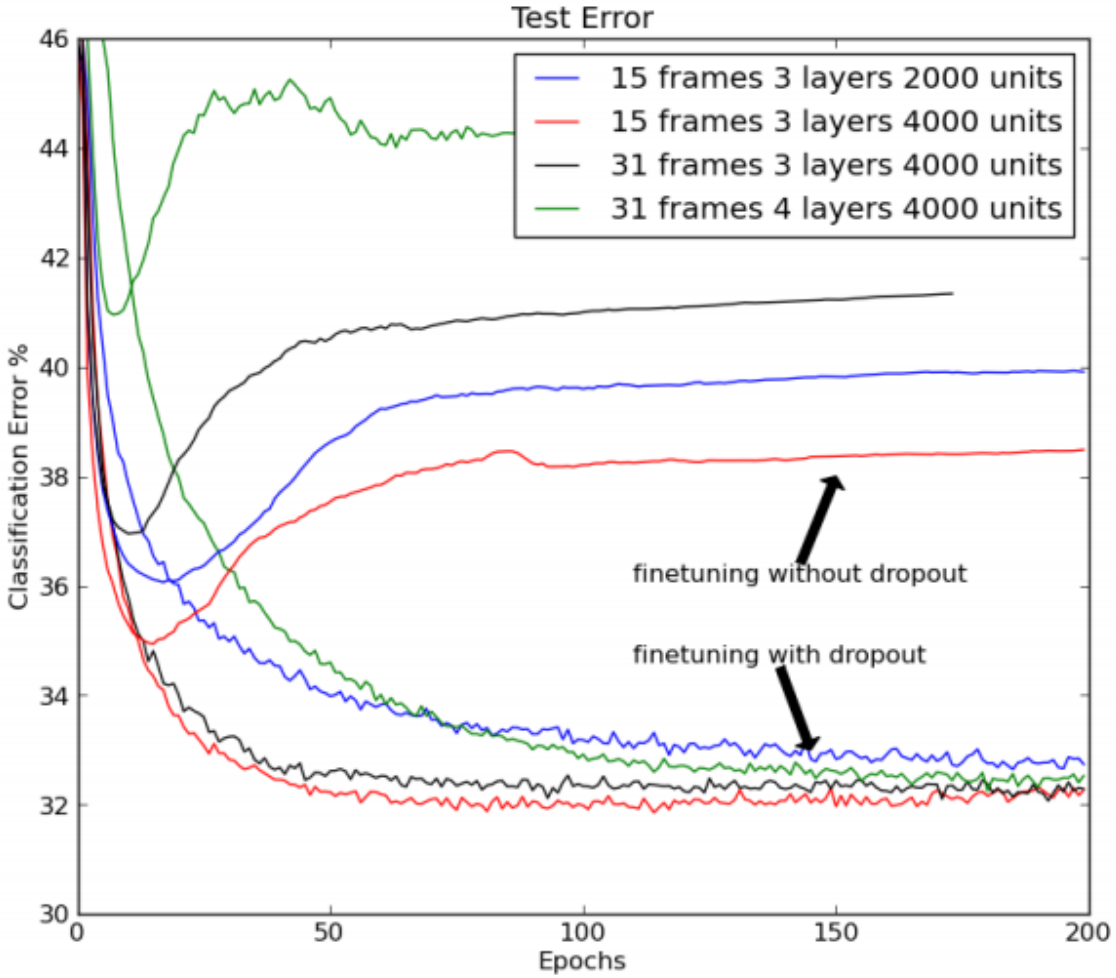
Tends to avoid overfitting

Has a model averaging effect

- Only some parameters trained at any step



Preventing overfitting: dropout training



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

Neural Networks

SUMMARY

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

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
- **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