Introduction to Modern AI
Week 4: Deep Learning

Gavin Hartnett

PRGS, Winter Quarter 2022

Topics we will cover this week

- Neural computation
    - Automatic differentiation
    - Deep learning libraries
    - Hardware accelerators
- Practical deep learning
    - Early stopping
    - Dropout
    - BatchNorm/LayerNorm
    - Improved SGD algorithms

Neural Computation

Automatic Differentiation

- In order to train NN, need to be able to compute the gradient of the loss function: $\nabla_{\boldsymbol{\theta}}$loss
- Can classify optimization algorithms according to how many derivatives they require
- Zeroth-order: just require function evaluation (no derivatives)
- First-order: require gradient
- Second-order: require second derivatives (Hessian)
- In principle, you could train a NN using any of these
- In practice, first-order methods are the only feasible ones
    - Zeroth-order: too inefficient
    - Second-order: too slow to compute Hessian, hard to fit into memory
    - First-order: just right!

Automatic Differentiation

- In order to train NN, need to be able to compute the gradient of the loss function: $\nabla_{\boldsymbol{\theta}}\text{loss}$
- Loss function depends on NN output, i.e. $f(\boldsymbol{x};\boldsymbol{\theta})$, and NN output depends on $\boldsymbol{\theta}$
- Problem: the gradient is sure to be a very complicated function, hard to write down and hard to evaluate
- There are 3 general approaches for getting a computer to compute derivatives/gradients

Automatic Differentiation

- Symbolic differentiation
  - Essentially operates via repeated application of chain rule
  - given a formula, returns a formula
  - $f(x) = x\sin(5x)$
  - $f'(x) = 5x\cos(5x) + \sin(5x)$
- Numerical differentiation
  - Uses finite difference formula(e) to compute derivative at a particular point (i.e., outputs a number, not a formula/function):

$$f'(x) \approx \frac{f(x+\Delta) - f(x_i)}{\Delta}$$

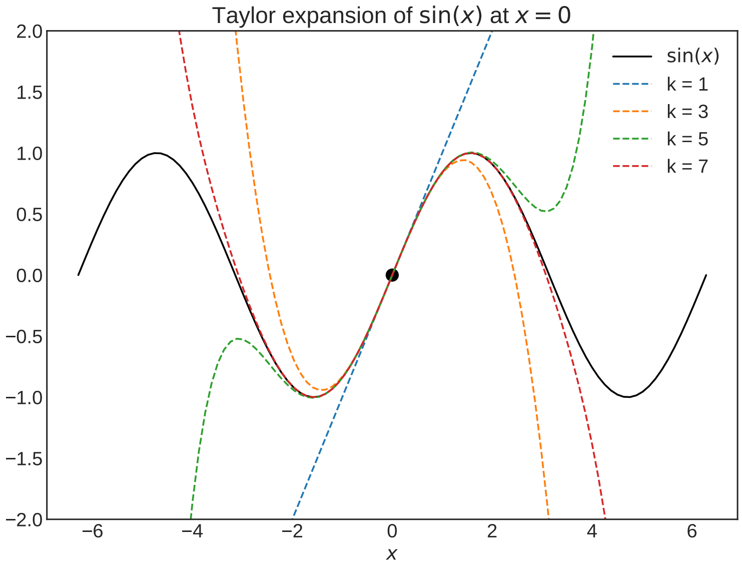
  - Becomes very expensive if x is a vector and not a scalar
- Automatic Differentiation
  - Essentially operates via repeated application of chain rule
  - Computation is NOT done symbolically, gradient at a point is returned (numerical vector as opposed to a formula)

Aside: Finite Difference

- To help motivate how awesome autodiff is, let's understand how bad one of the alternatives is
- Taylor expansion:

$$
\begin{aligned}
f(x) &= f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2} f''(x_0)(x - x_0)^2 + \ldots \\
&= \sum_{n=0}^{\infty} f^{(n)}(x_0)(x - x_0)^n
\end{aligned}
$$

Aside: Finite Difference



Aside: Finite Difference

- To help motivate how awesome autodiff is, let's understand how bad one of the alternatives is
- Taylor expansion:

$$\begin{aligned} f(x) &= f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2} f''(x_0)(x - x_0)^2 + \ldots \\ &= \sum_{n=0}^{\infty} f^{(n)}(x_0)(x - x_0)^n \end{aligned}$$

- If $h = x - x_0$ is small, than the higher order terms are even smaller $(h^2, h^3, \ldots)$
- linear approximation: $f(x) = f(x_0) + f'(x_0)h + \mathcal{O}(h^2)$, or

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} + \mathcal{O}(h)$$

Aside: Finite Difference

- This is for a x a scalar. For a vector the formula becomes

$$f(\boldsymbol{x}_0 + \boldsymbol{h}) \approx f(\boldsymbol{x}_0) + \boldsymbol{h}^T \nabla f(\boldsymbol{x}_0)$$

- We are free to pick the vector $\boldsymbol{h}$
- To order to isolate the i-th component of the gradient, pick $\boldsymbol{h}_i = h\,\boldsymbol{e}_i$, where $\boldsymbol{e}_i = (0, ..., \underbrace{1}_{i}, 0, ..., 0)$
- Then:

$$\nabla f(\boldsymbol{x_0})_i = \frac{f(\boldsymbol{x}_0 + h\,\boldsymbol{e}_i) - f(\boldsymbol{x}_0)}{h}$$

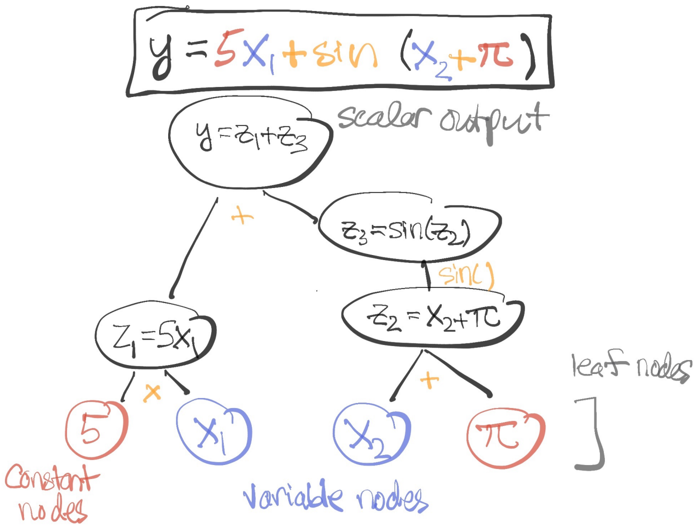
- Why is this not great
  - Takes d function calls to get full gradient vector
  - Result is only an approximation

Automatic Differentiation

Useful resources:

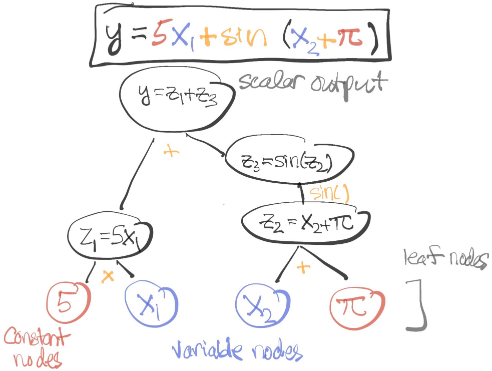
- Chris Olah's excellent blog post
- Ch 6.5 of Goodfellow, Bengio, Courville

Computational Graphs



Computational Graphs

- Nodes represent variables
- (directed) Edges represent operations
- Leaf nodes correspond to variables or constants
- Forward pass through graph corresponds to computing the output given the variables and constants



Computational Graphs: Reverse-Mode Differentiation

- The computational graph + chain rule helps us take gradients
- Example:

$$
\begin{aligned}
y &= 5x_1 + \sin(x_2 + \pi) \\
&= z_1 + \sin(z_2) \\
&= z_3
\end{aligned}
$$

- Chain rule yields:

$$
\begin{aligned}
\frac{\partial y}{\partial x_1} &= \frac{\partial z_3}{\partial x_1} \\
&= \frac{\partial z_3}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial z_3}{\partial z_2}\frac{\partial z_2}{\partial x_1} \\
&= \cancelto{1}{\frac{\partial z_3}{\partial z_1}} \; \cancelto{5}{\frac{\partial z_1}{\partial x_1}} + \frac{\partial z_3}{\partial z_2}\cancelto{0}{\frac{\partial z_2}{\partial x_1}} = 5
\end{aligned}
$$

Computational Graphs: Reverse-Mode Differentiation

- The computational graph + chain rule helps us take gradients
- Example:

$$\begin{aligned} y &= 5x_1 + \sin(x_2 + \pi) \\ &= z_1 + \sin(z_2) \\ &= z_3 \end{aligned}$$

- Chain rule yields:

$$\begin{aligned} \frac{\partial y}{\partial x_2} &= \frac{\partial z_3}{\partial x_2} \\ &= \frac{\partial z_3}{\partial z_1}\frac{\partial z_1}{\partial x_2} + \frac{\partial z_3}{\partial z_2}\frac{\partial z_2}{\partial x_2} \\ &= \frac{\partial z_3}{\partial z_1}\cancelto{0}{\frac{\partial z_1}{\partial x_2}} + \cancelto{\cos(z_2)}{\frac{\partial z_3}{\partial z_2}} \qquad \cancelto{1}{\frac{\partial z_2}{\partial x_2}} = \cos(z_2) = \cos(x_2 + \pi) \end{aligned}$$

Computational Graphs: Reverse-Mode Differentiation

- Final result:
$$\nabla_{\boldsymbol{x}} y = [5, \cos(x_2 + \pi)]$$
- To calculate the output, we did a forward pass (start at leaves, move to root)
- To calculate the gradient, we did a reverse or backwards pass (start with root and move to leaves)
- Used the fact that we know the derivatives of the simple operations used to build the computational graph $(+, \times, \sin()$, etc)
- For more complicated expressions, it is crucial to save intermediate results such as $\frac{\partial z_3}{\partial z_1}$
- Same basic approach works for symbolic and automatic differentiation, but implementation details are very different

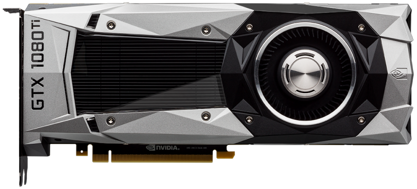
Deep Learning Libraries and Hardware Accelerators

Deep Learning Libraries

- Many deep learning libraries exist
- Most popular are open-source, well-supported, Python-based
    - PyTorch (Facebook)
    - Tensorflow (Google)
    - Jax (Google)
- We will use PyTorch: very popular, easy to use
- The primary advantages of these libraries are:
    - Have many built-in functions for deep neural networks
    - Autodiff capable
    - Allow for easy GPU support (do calculation on GPU instead of CPU)
- For HW 3 you will use PyTorch within Google Colab

Hardware Accelerators

- CPUs are general purpose computing devices
- Specialized hardware exists for various problems of interest (ASICs = Application Specific Integrated Circuits)
- Example: Bitcoin mining rigs
- Graphics Processing Units (GPUs) are specialized devices that excel at the linear algebra operations used in deep learning
- Tensor Processing Units (TPUs) are deep learning ASICs produced by Google



Source: https://www.nvidia.com/en-us/geforce/10-series/

PyTorch

Review PyTorch Tutorial (Week 4 Notebook)

Practical Deep Learning

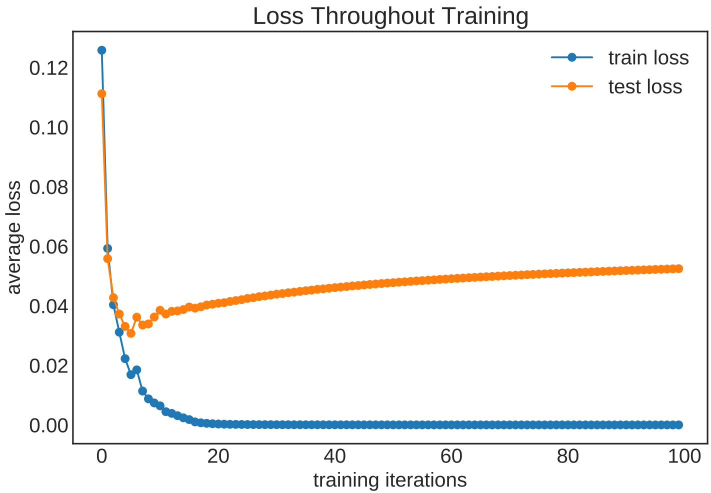
Practical Deep Learning

- Goal is to review some common practical methodologies (i.e., tricks) used in training deep neural networks
- Each of these has some nice motivation, but the use of these methods is driven by practical considerations (i.e., they just seem to work really well)

Early Stopping

- Early stopping is when you prematurely halt the training of a model before convergence
- Motivation 1: Save time - it might take a very long time for optimization to converge
- Motivation 2: Avoid overfitting - more the model is trained the greater the chance that overfitting occurs

Early Stopping



Early Stopping

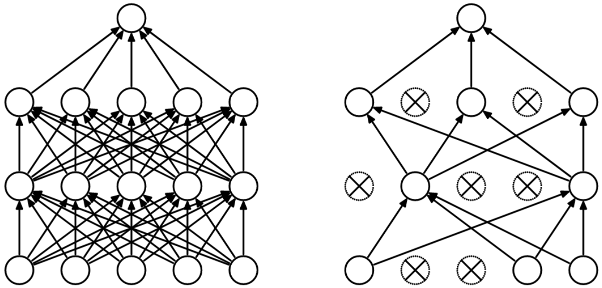
- Early stopping is when you prematurely halt the training of a model before convergence
- Motivation 1: Save time - it might take a very long time for optimization to converge
- Motivation 2: Avoid overfitting - more the model is trained the greater the chance that overfitting occurs
- Training time seems to affect the generalization gap in a qualitatively similar way as model capacity
- This makes sense - SGD takes time to go from a randomly initialized network to an overfitted one
- Can think of capacity as a function of training time

Dropout

- Ensemble methods are often quite useful
- For example, Bootstrap Aggregating (Bagging) allows the predictions of many different models to be combined, often results in lower variance
- It seems difficult to apply these ideas to NNs, however, because training even a single NN can be quite time consuming.
- Dropout is a very simple way to apply the spirit of ensemble methods to NNs
- Introduced in this excellent paper:
  Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." The journal of machine learning research 15.1 (2014): 1929-1958.

Dropout

Srivastava, Hinton, Krizhevsky, Sutskever and Salakhutdinov



(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Dropout

- During each training pass, drop out each unit with probability p
- During test pass, retain all units but multiply weights by p (why)
- In effect we are sampling networks from an ensemble of size $2^{N_{\text{units}}}$
- Motivation: break up “conspiracies” between neurons

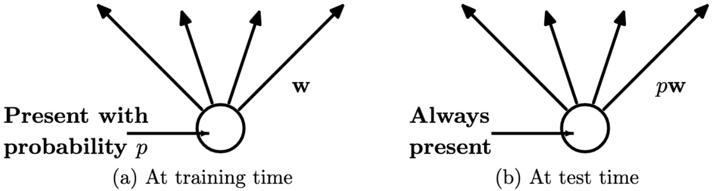


Figure 2: **Left**: A unit at training time that is present with probability p and is connected to units in the next layer with weights **w**. **Right**: At test time, the unit is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time.

BatchNorm

- BatchNorm is a layer-wise operation, meaning it acts on all the variables at layer ℓ
- Idea is to "normalize" the variables by removing the mean and rescaling to have unit variance

$$\mathrm{BatchNorm}(x)_a = \gamma_a \left(\frac{x_a - \mathbb{E}[x_a]}{\sqrt{\mathrm{Var}[x_a] + \epsilon}} \right) + \beta_a$$

- Here a indices the dimension (not sample)
- Expectation, variance are estimated using mini-batch samples
- γ, β are learnable vectors
- BatchNorm has the effect of improving the speed and stability of SGD
- LayerNorm: another approach with the same formula, only the expectation and variance are now computed across the feature dimension for each instance separately

Momentum

- Momentum is a simple modification of gradient descent (or SGD)
- Add a short-term "memory" to GD:

$$\boldsymbol{z}^{t+1} = \beta\, \boldsymbol{z}^{t} + \nabla f(\boldsymbol{w}^{t})$$
$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^{t} - \alpha\, \boldsymbol{z}^{t+1}$$
$$\text{with } \boldsymbol{z}^{0} = 0$$

- (α, β) are hyper-parameters
- When $\beta = 0$, recover GD
- Show Gabriel Goh's excellent distill article https://distill.pub/2017/momentum/
- Bottom line: momentum allows for larger learning rate α, and faster convergence