Intro to modern neural networks

COMP 4630 | Winter 2024

Charlotte Curtis

Overview

- More decisions when making a neural network
  - Weight initialization
  - Number of neurons and layers
  - Optimization algorithms
- References and suggested reading:
  - Scikit-learn book: Chapters 10-11
  - Deep Learning Book: Chapter 8
  - Understanding Deep Learning: Chapter 7

Revisiting Backpropagation

- For a network with l layers, the gradients of the loss function with respect to the weights in the last layer are given by:

$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial f^{(l)}} \frac{\partial f^{(l)}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial W^{(l)}}$$

  assuming that the output $\hat{y} = f^{(l)}(z^{(l)})$ is a function of layer l's input $z^{(l)} = W^{(l)} f^{(l-1)}(z^{(l-1)}) + b^{(l)}$.

- At layer $l-1$, the gradients are computed as:

$$\frac{\partial L}{\partial W^{(l-1)}} = \left(\frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial f^{(l)}} \frac{\partial f^{(l)}}{\partial z^{(l)}} \right) \frac{\partial z^{(l)}}{\partial f^{(l-1)}} \frac{\partial f^{(l-1)}}{\partial z^{(l-1)}} \frac{\partial z^{(l-1)}}{\partial W^{(l-1)}}$$

- At layer $l-2$, this becomes:

$$\frac{\partial L}{\partial W^{(l-2)}} = \left(\frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial f^{(l)}} \frac{\partial f^{(l)}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial f^{(l-1)}} \frac{\partial f^{(l-1)}}{\partial z^{(l-1)}} \right) \frac{\partial z^{(l-1)}}{\partial f^{(l-2)}} \frac{\partial f^{(l-2)}}{\partial z^{(l-2)}} \frac{\partial z^{(l-2)}}{\partial W^{(l-2)}}$$

- And so on, until we reach the first layer.
- We are **recursively** applying the chain rule and re-using the gradients computed at the previous layer
- This is great for computational efficiency, but it can also lead to **vanishing** or **exploding gradients**

Vanishing and Exploding Gradients

- **Vanishing/exploding gradients** are where the gradients become near zero or near infinity as they are propagated back through the network
- Particularly problematic for **recurrent** neural networks, where the same weights are multiplied by themselves repeatedly
- Also a problem for very deep networks, and part of the reason that deep learning was not popular until the 2010s
- ❓ What changed?

Consider the variance

- At the input layer, $Z^{(0)} = W^{(0)}X + b^{(0)}$, and X has some variance σ^2
- Assume $W^{(0)}$ and $b^{(0)}$ are initialized to 0
- 🧮 What is the variance of $Z^{(0)}$?
- What about after the activation function $f^{(0)}(Z^{(0)})$?

Initialization strategies

- In 2010, Glorot and Bengio proposed the **Xavier** initialization for a layer with m inputs and n outputs:

$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$$

- Goal is to preserve the variance of the input and output in both directions
- Similar to LeCun initialization, and apparently an overlooked feature of networks from the 1990s

Initialization for ReLU

- Glorot initialization was derived under the assumption of **linear** activation functions (even though they knew this wasn't the case)
- In 2015, He et al. proposed the **He** initialization specifically for ReLU activations:

$$W_{i,j} \sim N\left(0, \sqrt{\frac{2}{m}}\right)$$

- This can be defined in Keras as `kernel_initializer='he_normal'` or `kernel_initializer='he_uniform'`
- The choice of normal vs uniform is apparently not very important

Batch normalization

- Also in 2015, Ioffe and Szegedy proposed **batch normalization** as a way to mitigate vanishing/exploding gradients
- This is simply a normalization **at each layer**, shifting and scaling the inputs to have a mean of 0 and a variance of 1 (across the batch)
- A **moving average** of the mean and variance is maintained **during training**, and used for normalization during inference
- It also ends up acting as **regularization**, magic!
- ❓ Why wouldn't you want to use batch normalization?

RELU and its variants

- In early works, the sigmoid or tanh functions were popular
- Both have a small range of non-zero gradients
- ReLU has a stable gradient for positive inputs, but can lead to the **dying ReLU** problem whereby certain neurons are "turned off"
- ❓ How can we prevent dying ReLUs?

> *Note: this may not be a problem, and ReLU is cheap. Don't optimize prematurely unless you're seeing lots of "dead" neurons.*

Number of neurons and layers

- Number of neurons in the **input layer** is defined by number of features
- Number of neurons in the **output layer** is defined by prediction task
- In between is a **design choice**
- Common early choice was a pyramid shape, but it turns out that a stack of layers with the same number of neurons works well too
- Deeper networks can solve more complex problems with the same number of total parameters, but are also prone to vanishing/exploding gradients
- Ultimately a hyperparameter to be tuned

Optimization algorithms: variations on gradient descent

- Gradient descent takes small regular steps, constant or otherwise
- Many variations exist! For example, **momentum** keeps track of the previously computed gradient and uses it to inform the new step:

$$\mathbf{m} = \beta\mathbf{m} - \eta\nabla_{\mathbf{W}}J(\mathbf{W})$$
$$\mathbf{W} = \mathbf{W} + \mathbf{m}$$

  where β is a hyperparameter between 0 and 1
- Adaptive moment estimation (**Adam**) is a popular choice that adds on an exponentially decaying average of the squared gradients

The Adam optimizer

- Keeps track of first ($\mathbf{s}$) and second ($\mathbf{r}$) moments of the gradient, with two exponential decay terms ρ_1 and ρ_2
- At each time step, the update is now:

$$\mathbf{s} = \rho_1 \mathbf{s} + (1 - \rho_1) \nabla_{\mathbf{W}} J(\mathbf{W})$$
$$\mathbf{r} = \rho_2 \mathbf{r} + (1 - \rho_2) \nabla_{\mathbf{W}} J(\mathbf{W})^2$$
$$\mathbf{W} = \mathbf{W} - \eta \frac{\mathbf{s}}{\sqrt{\mathbf{r}} + \epsilon}$$

- ρ_1 and ρ_2 are typically 0.9 and 0.999, respectively

Regularization via dropout

- **Dropout** is a regularization technique that randomly sets a fraction of the neurons to zero during training
- During each training pass, a neuron has a p probability of being dropped
- Similar to training an ensemble of models then bagging
- Helps to prevent overfitting, but can slow down training
- Typical values: 0.5 for hidden layers, 0.2 for input layers

Choices for starting

From the Deep Learning Book, section 11.2:

- "Depending on the complexity of your problem, you may even want to begin without using deep learning."
- Tabular data: fully connected, images: convolutional, sequences: recurrent*
- ReLU or variants, with He initialization
- SGD or Adam, add batch normalization if unstable
- Use some kind of regularization, such as dropout

*This book was written before transformer models

Implementation time