2.2 Training Using Backpropagation and Gradient Descent

Training Using Backpropagation and Gradient Descent

Training a neural network means finding the right values for its internal parameters so that the network can make accurate predictions on unseen data.

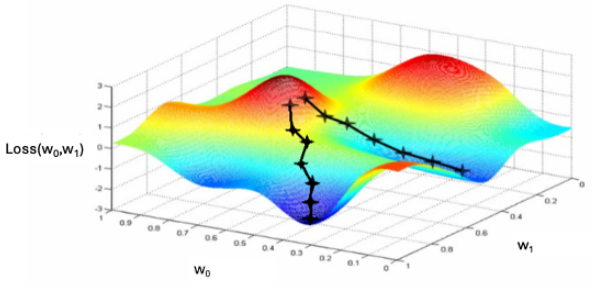
At a high level, training involves:

- **Forward Pass:** Loading the sample into the input layer, letting it propagate through each layer, and calculating the loss at the output.
- **Backpropagation:** Computing from output layer back to the input how much each weight contributed to the loss.
- **Gradient Descent:** Updating the weights to move in the direction of smaller loss.

This is repeated across multiple **epochs**, where one epoch means a full pass through all of the training data.

The forward pass was described in some detail last week. Understanding the remaining two steps is essential in understanding how to build deep learning models.

Backward Pass: Computing Gradients via Backpropagation



Backpropagation is an algorithm for exploring a P -dimensional space $\mathbb{R}^P$ (where P is the number of parameters) for the point with the minimum loss. In two dimensions, we can

visualize this as starting at some point in a landscape and traveling downhill to find the lowest "valley" (Figure 3).

In detail, what backpropagation does is calculate the **partial derivative** for each parameter w :

$$\frac{\partial L}{\partial \mathbf{w}}$$

This is the instantaneous rate of change (i.e., the "slope along the w-axis") of the loss as you vary that one parameter w , holding all others fixed.

Figure 3: *Gradient Descent Landscape*

Source: Suryansh, S. (2018, March 12). *Gradient descent: All you need to know.* (https://medium.com/hackernoon/gradient-descent-aynk-7cbe95a778da) *Medium*.

Gradient Descent: Updating the Weights

Once we have the gradients from backpropagation, we use them to update the weights using **gradient descent.** The idea is simple: adjust each weight in the direction that reduces the loss (i.e., "downhill").

The basic **update rule** is:

$$\mathbf{w} = \mathbf{w} - \eta \frac{\partial L}{\partial \mathbf{w}}$$

where η is the learning rate specifying how big a "step" to take downhill in the loss landscape. Choosing an appropriate learning rate is crucial: Make it too small, and training is slow and may not reach an optimal minimum point; make it too large, and training becomes unstable, and you may jump over a local minimum and miss it completely!

Optimizing Gradient Descent

Over time, a variety of improvements to the basic gradient descent algorithm have been developed. (These will be discussed in the upcoming coding video.)

Common Optimizations

- **Mini-batch Gradient Descent**: Divides the dataset into mini-batches of 8 or 16 or 32 samples and updates weights after calculating gradients from all samples in the batch.
- **SGD (Stochastic Gradient Descent)**: Updates weights using a small random batch of training data instead of the whole dataset.
- **Momentum**: Adds a velocity term so updates can accelerate in consistent directions and dampen oscillations. This is an important optimization that can avoid local minima. Think

of a ball rolling downhill: its momentum makes sure that it doesn't stop as soon as it must go up a tiny incline.

- **RMSProp**: Scales learning rates adaptively for each parameter using a moving average of squared gradients.
- **Adam**: Combines momentum and RMSProp for fast and robust training. Adam is one of the most popular optimizers and a common default choice.
- **Learning Rate Scheduling:** As training progresses, it often helps to reduce the learning rate to allow the model to settle into a good local minimum. Some common options here include:
    - **Step decay**: Reduce the learning rate every k epochs.
    - **Exponential decay**: Multiply the learning rate by a fixed factor (e.g., 0.98) at each step.
    - **Warm-up and cool-down**: Start with a small rate, increase it, and then decrease it again.

Schedulers can make training more stable and prevent the model from getting stuck in poor local minima.

Conclusion

Understanding how neural networks perform classification and regression tasks and learn through backpropagation and gradient descent lays the groundwork for building powerful models. Next week, we'll shift our focus to practical strategies for improving training performance — such as weight initialization, dropout, and early stopping — while also addressing the challenges of underfitting and overfitting. As always, our goal will be to find models that not only perform well on training data but also generalize effectively to new, unseen examples.