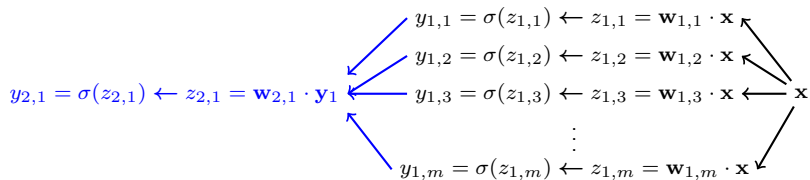


Neural Networks: Training and SGD

Spring 2021

Completing our matrix formalism



Completing our matrix formalism

Diagram illustrating the forward pass of a neural network layer:

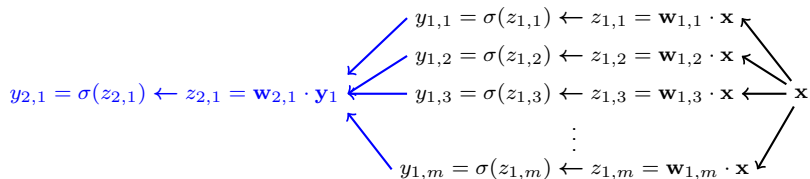
$$\begin{aligned} y_{1,1} &= \sigma(z_{1,1}) \leftarrow z_{1,1} = \mathbf{w}_{1,1} \cdot \mathbf{x} \\ y_{1,2} &= \sigma(z_{1,2}) \leftarrow z_{1,2} = \mathbf{w}_{1,2} \cdot \mathbf{x} \\ y_{1,3} &= \sigma(z_{1,3}) \leftarrow z_{1,3} = \mathbf{w}_{1,3} \cdot \mathbf{x} \\ &\vdots \\ y_{1,m} &= \sigma(z_{1,m}) \leftarrow z_{1,m} = \mathbf{w}_{1,m} \cdot \mathbf{x} \end{aligned}$$

The output vector \mathbf{y}_1 is then used to compute the final output y_2 :

$$y_{2,1} = \sigma(z_{2,1}) \leftarrow z_{2,1} = \mathbf{w}_{2,1} \cdot \mathbf{y}_1$$

$$\mathbf{y}_2 = \boldsymbol{\sigma}(\mathbf{z}_2) \quad \leftarrow \quad \mathbf{z}_2 = \mathbf{W}_2 \mathbf{y}_1 \quad \leftarrow \quad \mathbf{y}_1 = \boldsymbol{\sigma}(\mathbf{z}_1) \quad \leftarrow \quad \mathbf{z}_1 = \mathbf{W}_1 \mathbf{x}$$

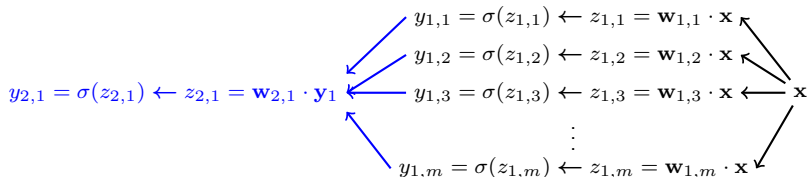
Completing our matrix formalism



$$\mathbf{y}_2 = \sigma(\mathbf{z}_2) \quad \leftarrow \quad \mathbf{z}_2 = \mathbf{W}_2 \mathbf{y}_1 \quad \leftarrow \quad \mathbf{y}_1 = \sigma(\mathbf{z}_1) \quad \leftarrow \quad \mathbf{z}_1 = \mathbf{W}_1 \mathbf{x}$$

$$\mathbf{y}_2 = \sigma(\mathbf{W}_2(\sigma(\mathbf{W}_1 \mathbf{x})))$$

Completing our matrix formalism



$$\mathbf{y}_2 = \sigma(\mathbf{z}_2) \quad \leftarrow \quad \mathbf{z}_2 = \mathbf{W}_2 \mathbf{y}_1 \quad \leftarrow \quad \mathbf{y}_1 = \sigma(\mathbf{z}_1) \quad \leftarrow \quad \mathbf{z}_1 = \mathbf{W}_1 \mathbf{x}$$

$$\mathbf{y}_2 = \sigma(\mathbf{W}_2(\sigma(\mathbf{W}_1 \mathbf{x})))$$

In general, a network with d layers is

$$\mathbf{y}_d = \sigma(\mathbf{W}_d(\sigma(\mathbf{W}_{d-1}(\cdots \sigma(\mathbf{W}_2(\sigma(\mathbf{W}_1 \mathbf{x}))))))).$$

Training neural networks

We need two things to train neural networks:

1. loss function L that measures how well the output of the final layer (y_d) compared with known training data.
2. A set of training data $(\mathbf{x}_i, y_i^{\text{true}})$.

Training neural networks

We need two things to train neural networks:

1. loss function L that measures how well the output of the final layer (y_d) compared with known training data.
2. A set of training data $(\mathbf{x}_i, y_i^{\text{true}})$.

Let's assume we have some loss function L that measures how well the output of the final layer (y_d) compared with known training data. For example, we could use quadratic loss

$$L_i = (y_{d,i} - y_i^{\text{true}})^2$$

Training neural networks

We need two things to train neural networks:

1. loss function L that measures how well the output of the final layer (y_d) compared with known training data.
2. A set of training data $(\mathbf{x}_i, y_i^{\text{true}})$.

Let's assume we have some loss function L that measures how well the output of the final layer (y_d) compared with known training data. For example, we could use quadratic loss

$$L_i = (y_{d,i} - y_i^{\text{true}})^2$$

The loss function depends on all the weights in the network, i.e.

$L_i(\mathbf{W}_d, \mathbf{W}_{d-1}, \dots, \mathbf{W}_2, \mathbf{W}_1)$. We will abbreviate this as simply $L_i(\mathbf{W})$.

Gradient descent

We can use a gradient descent scheme to find the weights. Given an initial (random) set of weights $\mathbf{W}^{(0)}$:

1. Calculate the gradient of the *total loss*

$$\mathbf{g}(\mathbf{W}) = \sum_{i=1}^N \frac{\partial L_i(\mathbf{W})}{\partial \mathbf{W}}.$$

2. Update the weights using

$$\mathbf{W}^{(1)} = \mathbf{W}^{(0)} - \alpha \mathbf{g}(\mathbf{W}^{(0)}).$$

3. Repeat.

Gradient descent

We can use a gradient descent scheme to find the weights. Given an initial (random) set of weights $\mathbf{W}^{(0)}$:

1. Calculate the gradient of the *total loss*

$$\mathbf{g}(\mathbf{W}) = \sum_{i=1}^N \frac{\partial L_i(\mathbf{W})}{\partial \mathbf{W}}.$$

2. Update the weights using

$$\mathbf{W}^{(1)} = \mathbf{W}^{(0)} - \alpha \mathbf{g}(\mathbf{W}^{(0)}).$$

3. Repeat.

The problem is the gradient calculation.

- ▶ The gradient has one entry for each parameter, of which there are thousands!
- ▶ These computations are repeated over all N points in the dataset *for each iteration*!

Stochastic Gradient Descent (SGD)

- ▶ For neural network training we use an alternative to gradient descent called *Stochastic Gradient Descent*, or SGD.
- ▶ SGD alternates between forward and backward passes through the model.
- ▶ SGD updates parameters using the loss for a single point.
- ▶ A single point provides a noisy, or *stochastic* approximation to the true loss gradient, but it is far more efficient.

The SGD algorithm

1. Select a single training point $(\mathbf{x}_i, y_i^{\text{true}})$.
2. Make a **forward pass** through the model to compute the output of the neural network

$$y_{d,i} = \sigma(\mathbf{W}_d(\sigma(\mathbf{W}_{d-1}(\cdots \sigma(\mathbf{W}_2(\sigma(\mathbf{W}_1 \mathbf{x}_i))))))).$$

3. Compute the loss for this single prediction

$$L_i = (y_{d,i} - y_i^{\text{true}})^2.$$

4. Calculate the gradient at this point and the current weights.
5. Update the weights using

$$\mathbf{W}^{(k+1)} = \mathbf{W}^{(k)} - \alpha \mathbf{g}(\mathbf{W}^{(k)}).$$

This completes the **backward pass**.

6. Repeat for the next training point.

The SGD algorithm (simplified)

1. Select a single training point $(\mathbf{x}_i, y_i^{\text{true}})$.
2. Make a **forward pass** through the model to compute the output of the neural network

$$y_{d,i} \leftarrow \text{neural network} \leftarrow \mathbf{x}_i$$

3. Compute the loss for this single prediction

$$L_i \leftarrow (y_{d,i}, y_i^{\text{true}})$$

4. Calculate the gradient at this point and the current weights.
5. Update the weights using

$$\frac{\partial L_i}{\partial \mathbf{W}} \rightarrow \mathbf{W}$$

This completes the **backward pass**.

6. Repeat for the next training point.

Implementing SGD

- ▶ Like gradient descent, SGD requires a step size hyperparameter (α).
- ▶ One pass through the entire training set is called an *epoch*.
- ▶ One epoch is not enough; often thousands are needed. The order of the training data is randomized between epochs.
- ▶ Some algorithms form *minibatches* by averaging a small number of training samples before updating the weights.

Why SGD?

- ▶ SGD is more efficient. Most datasets contain more points than are necessary to estimate the average, so computation is wasted.
- ▶ SGD is stochastic, just like the data. The stochasticity is a form of regularization.
- ▶ It just works, even if we don't understand why.

How do neural networks learn so well?

Neural network training *should* get stuck in local optima, but it doesn't. There are three theories why:

1. Lottery ticket theory: A good network is already somewhere in the randomly-initialized network.
2. There are no true local optima in high-dimensional spaces.
3. The noise in SGD allows training to “escape” local optima.

What else can we do to improve training?

1. **Regularize.** Neural networks can memorize anything, so we need to regularize them aggressively.
2. **Boosting.** Train in several rounds, weighting the loss function toward points that are not predicted correctly.
3. **Bagging.** Split the training data into parts and train a separate model on each part. Average the predictions of all the models.
4. **Experiment.** Change number and size of layers, hyperparameters, optimizers, batching, and regularization.