ML and Opt Lect 19

- Backprop for MLPs
- Automatic differentiation & ML frameworks
- Autoencoders
- Example of using Pytorch to learn a 2-layer classifier for Fashion MNIST
Let's give the relevant backprop formulas for the MLP case:

\[ o^l = \sigma(a^l) = \sigma(w^{l, l-1} o^{l-1} + b^l) \]

\[ o^{l+1} = \sigma(a^{l+1}) = \sigma(w^{l+1, l} o^l + b^{l+1}) \]

We have, by the chain rule,

\[ J_{o^{l+1}} (o^l) = \text{diag}(\sigma'(a^{l+1})) \cdot w^{l+1} \in \mathbb{R}^{n_{l+1} \times n_l} \]

\[ J_{o^l} (b^l) = \text{diag}(\sigma'(a^l)) \in \mathbb{R}^{n_l \times n_l} \]

and

\[ J_{o^l} (w^l) \in \mathbb{R}^{n_l \times (n_l \times n_{l-1})} \text{ satisfies} \]

\[ \left[ J_{o^l} (w^l) \right]_{i, :} = \left[ \frac{\partial (o^l)_i}{\partial w^l} \right] \in \mathbb{R}^{n_l \times n_{l-1}} \]

\[ = \text{diag}(\sigma'(a^l)) e_i (o^{l-1})^T \]
so we can show that for any vector $v \in \mathbb{R}^n$, 

$$J_{\omega}(\omega^T)v = \text{diag}(\sigma'(a^T))v(a^T\gamma) \in \mathbb{R}^{n \times n - 1}$$
Putting these pieces together

**Backprop for MLPs**

\[
\text{temp} \leftarrow \nabla_{o_l} f
\]
\[
\nabla_{b_l} f = \text{diag}(o'(a^L)) \text{temp} + \lambda \nabla_{b_l} R_L
\]
\[
\nabla_{w_l} f = \text{diag}(o'(a^L)) \text{temp} (o^{L-1})^T + \lambda \nabla_{w_l} R_L
\]

for \( l = L-1, \ldots, 1 \)

\[
\text{temp} \leftarrow (w^{L+1})^T \text{diag}(o'(a^{L+1})) \text{temp}
\]
\[
\nabla_{b_l} f \leftarrow \text{diag}(o'(a^l)) \text{temp} + \lambda \nabla_{b_l} R_L
\]
\[
\nabla_{w_l} f \leftarrow \text{diag}(o'(a^l)) \text{temp} (o^{l-1})^T + \lambda \nabla_{w_l} R_L
\]

Notes:
(i) in practice we cache the preactivations \( a^l \) during the forward pass
(ii) cost of forward & backward passes similar
(iii) costs of passes dominated by lin algebra ops (most vec prods)
BP is a core computational tool for working with NNs. Become familiar with it—we will see that it gives rise to some important optimization phenomena when training NNs.

- We can implement BP for any computational DAG with (sub)differentiable neurons. In practice doing this by hand is tedious, error-prone, and requires careful optimizations for efficiency.

- Now we have available several high-quality ML frameworks whose main focus is to automate BP as much as possible: you specify how \( f(x) \) is computed in a forward pass, then they deduce the backward pass using autodifferentiation. They also optimize the computations and support GPUs for much faster computation.
Common ML Frameworks:

- **Pytorch** (we use in this class; more popular now in research)
  2016 Facebook Research

- **Tensorflow** (used the last time course was taught; more used in production)
  2015 Google Brain
Autoencoders

- Nonlinear generalization of PCA & SVD

- Typically autoencoders are used to create useful features for a downstream task, in an unsupervised way.

Goal: learn a (compressed) representation that does a good job of reconstructing the input.
$$\text{Ex } d_2 \text{- loss } l(u,v) = \| u - v \|_2^2$$

and $\sigma(x) = x$

$$\hat{y}(x) = o^2 = \sigma(\omega^2 o^1 + b^2)$$
$$= \sigma(\omega^2 \sigma(\omega^1 x + b^1) + b^2)$$
$$= \omega^2 \omega^1 x + \omega^2 b^1 + b^2$$

Fix $b^1 = b^2 = 0$ then

$$\hat{y}(x) = \omega^2 \omega^1 x$$

where $p$

so the autoencoder learns $\omega^2, \omega^1$

to minimize

$$f(\omega^1, \omega^2) = \frac{1}{n} \sum_{i=1}^{n} \| \hat{y}(x_i) - x_i \|_2^2$$

$$= \frac{1}{n} \sum_{i=1}^{n} \| \omega^1 \omega^2 x_i - x_i \|_2^2$$
$$= \frac{1}{n} \| \omega^1 \omega^2 X - X \|_F^2$$

where $X = [x_1, \ldots, x_n]$
so fitting an autoencoder with \( \ell(u, v) = \|u - v\|^2 \) and \( b^1 = b^2 = 0 \) is equivalent to solving

\[
\omega^2, \omega' = \arg\min_{\omega^2 \in \mathbb{R}^{n \times d}} \| \omega^2 \omega' x - x \|_F^2
\]

this is the SVD!

Clearly autoencoders are a more flexible method of dim reduction than SVD because one can take a nonlinear \( \sigma \) and nonzero biases.
Deep autoencoders: