TRAINING NEURAL NETWORKS

Recall logistic regression:
1. Define prediction function $f_{\vec{w},b}$
2. Specify loss and cost fn. $J(\vec{w},b)$
3. Train on data by minimizing cost (using e.g. gradient descent)

For NN w/ TF: (w/ activation fn)
1. Build Sequential model of Dense layers
2. Specify loss fn to model.compile
3. Train on data w/ model.fit(X, Y, #epochs)

Activation Functions

So far we've built a NN by stringing together a bunch of log. reg. models (since activation fn = sigmoid). But we can allow a larger range of activation values w/ diff. fns, e.g. "ReLU" where $g(z) = \max(0, z)$. Linear also sometimes used (aka "none"), softmax.
(And others exist too.)

How to choose?
- Output layer:
  - for binary classification, Sigmoid is natural (b/c network learns to predict probabilities)
  - for regression, linear is natural, since this permits full $\mathbb{R}$ output (which is natural for regression)
  - for multiple classes, ReLU is natural: nonzero but multiple values.

- Hidden layers — ReLU is most common. Most efficient to compute; gradient descent is faster (bc fn is less "flat")

Why activation functions?
→ If all linear, this is no different from linear regression. (Recall linear space is closed under function composition (f∘g is linear for f,g linear)
→ Even w/ logistic output activation and linear activation for hidden layers, this is equiv. to logistic reg.

ReLU
→ enables models to "stitch together" linear segments to model complex nonlinear functions
→ by tuning $\vec{w}, b$ for each segment we can keep it "quiet" until needed in a specific range.

MULTICLASS CLASSIFICATION

- N possible classes, need to predict which
- Learn decision boundary that divides the space into regions, not just halves

Soft max

- generalization of logistic regression

For classes $\{1, 2, \ldots, n\}$,

$$a_i = \frac{e^{z_i}}{e^{z_1} + e^{z_2} + \ldots + e^{z_i} + \ldots + e^{z_n}} \quad \text{for } i \in \{1, \ldots, n\}$$

$$= P(Y = i \mid X = \vec{x})$$

and $z_i = \vec{w}_i \cdot \vec{x} + b_i$

Note that $\sum_{i=1}^{n} a_i = 1$

(For $n=2$, this reduces to the logistic regression binary classification model)

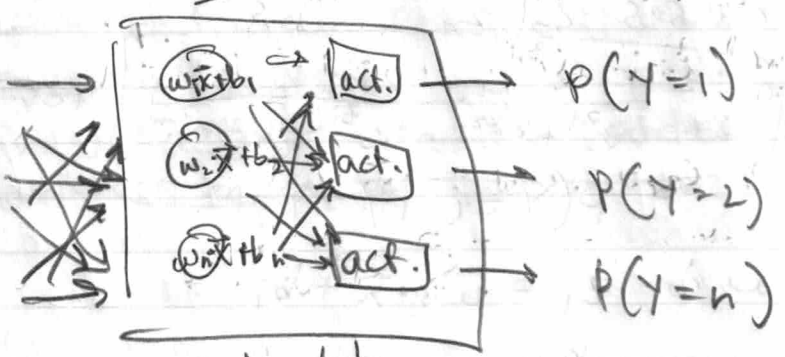
$$\text{Loss}(a_1, \ldots, a_n, y) = \begin{cases} -\log a_1 & \text{for } y=1 \\ -\log a_i & \text{for } y=i \\ -\log a_n & \text{for } y=n \end{cases}$$

For $y=j$, this makes $\text{Loss} = -\log j$, so that for $j \to 0 \Rightarrow \text{loss} \to \infty$ and $j \to 1 \Rightarrow \text{loss} \to 0$

Neural Network w/ Softmax

Add "softmax" output layer w/ n nodes for n output classes s.t. activation fn. for node i is a_i.

Note that this layer is different since the activation of each node requires the pre-activation output from all nodes in that layer.



Improvement in practice: For Log. Reg.: avoid numerical roundoff errors by specifying `from_logits=True` in the loss = Binary Cross Entropy arg to model.compile For Softmax: specify to loss = Sparse Categorical Cross Entropy in model.compile and use linear activation in output layer. Then for predictions do tf.nn.sigmoid/softmax respectively.

Multi-label

- single input, multiple outputs possible
- e.g. image has {car, person, bus} i.e. all of them

Option 1: Separate NNs to detect each

Option 2: Output layer outputs n probabilities, i.e. has n nodes.

ADVANCED OPTIMIZATION

Recall Gradient Descent:

$$w_j \leftarrow w_j - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b)$$

(α: learning rate)

Adam algorithm adapts α based on how G.D. is proceeding: same dir, faster; moving a lot, smaller.

→ uses $\vec{\alpha} = (\alpha_1, \ldots, \alpha_n)$ vector of learning rates since this is coord/vaker-wise.

(Details are beyond the scope of the course)

→ In TF specify Adam to model.compile

ADDITIONAL LAYER TYPES

- Dense layer: standard, combines all outputs from previous layer
- Convolutional layer: each neuron only considers a subset of the outputs from prev. layer
  - can help avoid overfitting
  - can require less data for training

Example: prediction based on EKG data might use a convolutional layer to consider sliding windows of the input data (w/ further conv. layers to combine into larger "windows") — predicting e.g. heart disease

Creating/combining layer types is a major focus of NN research.

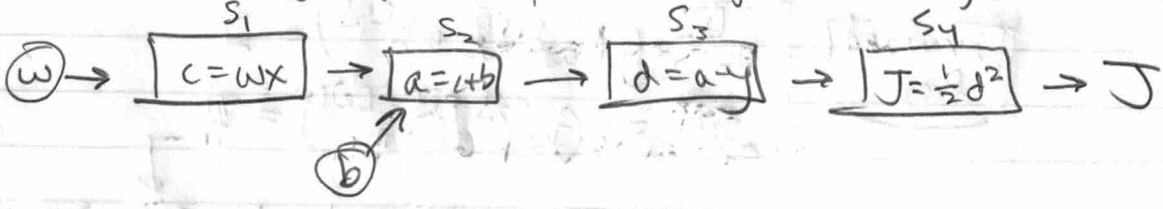
BACK PROPAGATION

Computation Graph breaks a larger computation, e.g. cost fn. into a DAG of smaller steps. For example, can model forward prop. w/ a computation graph.

→ How to compute the derivative of J:
Take derivatives moving backwards in the graph! Plugging in values from "later" steps and applying the chain rule.

Small example. Consider linear regression model: $x \to \boxed{\bigcirc}_{w,b} \to a = wx + b$ with loss fn. $J(w,b) = \frac{1}{2}(a-y)^2$

The (a) comp. graph is, for a given (x,y):



Using back-propagation to compute J:

$$s_4: \frac{\partial J}{\partial d} = d$$
$$s_3: \frac{\partial d}{\partial a} = 1$$
$$s_2: \frac{\partial a}{\partial b} = 1, \frac{\partial a}{\partial c} = 1$$
$$s_1: \frac{\partial c}{\partial w} = x$$

(via chain rule)

$$\to \nabla J = \begin{bmatrix} \frac{\partial J}{\partial w} \\ \frac{\partial J}{\partial b} \end{bmatrix} = \begin{bmatrix} \frac{\partial J}{\partial d}\frac{\partial d}{\partial a}\frac{\partial a}{\partial c}\frac{\partial c}{\partial w} \\ \frac{\partial J}{\partial d}\frac{\partial d}{\partial a}\frac{\partial a}{\partial b} \end{bmatrix} = \begin{bmatrix} (d)(1)(1)(x) \\ (d)(1)(1) \end{bmatrix} = \begin{bmatrix} xd \\ d \end{bmatrix}$$

For given (w,b,x,y) this is (using graph above) $d = wx + b - y$

e.g. $w=2, b=8, x=-3, y=2 \Rightarrow \underline{d=2}$ so

$$\nabla J = \begin{bmatrix} -6 \\ 2 \end{bmatrix}.$$

(N nodes w/ P params: N+P instead of NP steps.)

Why use this? Efficient derivative calculation — this is essentially dynamic programming since ∇J requires computing all the transitive partial derivatives combined. Do it bottom-up! linear time!

Larger example

$x=1,\ y=5,\ \vec{w}=\begin{bmatrix}2\\3\end{bmatrix}\ \vec{b}=\begin{bmatrix}0\\1\end{bmatrix}\ \ g(z)=\max(0,z)$

$\vec{x} \to \boxed{\bigcirc} \to a^{[1]} \to \boxed{\bigcirc} \to a^{[2]}$

Forward Prop

$a^{[1]} = g(w^{[1]}x + b^{[1]}) = 2$

$a^{[2]} = g(w^{[2]}a^{[1]} + b^{[2]}) = 7$

So loss $J(\vec{w},\vec{b})$ at $(x,y) = \frac{1}{2}(a^{[2]}-y)^2$

$= \frac{1}{2}(7-5)^2$

$= 2$

As a comp-graph:

$t^{[1]} = w^{[1]}x \to z^{[1]} = t^{[1]} + b^{[1]} \to a^{[1]} = g(z^{[1]}) \to$

$\hookrightarrow t^{[2]} = w^{[2]}a^{[1]} \to z^{[2]} = t^{[2]} + b^{[2]} \to a^{[2]} = g(z^{[2]})$

(values: $t^{[1]}=2$, $z^{[1]}=2$, $a^{[1]}=2$, $t^{[2]}=6$, $z^{[2]}=7$, $a^{[2]}=7$, $J=2$)

So $\frac{\partial J}{\partial a^{[2]}} = (a^{[2]}-y) = 2$ $\quad \frac{\partial a^{[1]}}{\partial z^{[1]}} = 1$ $\quad J = \frac{1}{2}(a^{[2]}-y)^2 = 2$

$\frac{\partial a^{[2]}}{\partial z^{[2]}} = \frac{\partial g}{\partial z^{[2]}}$ $\quad \frac{\partial z^{[2]}}{\partial t^{[2]}} = 1$ (for $z^{[2]} > 0$) $\quad \frac{\partial t^{[2]}}{\partial a^{[1]}} = w^{[2]} = 3$

$\frac{\partial z^{[2]}}{\partial b^{[2]}} = 1$ $\quad \frac{\partial z^{[2]}}{\partial t^{[2]}} = 1$ $\quad \frac{\partial t^{[2]}}{\partial w^{[2]}} = a^{[1]} = 2$

$\frac{\partial z^{[1]}}{\partial b^{[1]}} = 1$ $\quad \frac{\partial z^{[1]}}{\partial t^{[1]}} = 1$

$\frac{\partial t^{[1]}}{\partial w^{[1]}} = x$

Before TF/PyTorch you had to do this differentiation manually! "AutoDiff"

$\nabla J = \left(\frac{\partial J}{\partial w^{[1]}}, \frac{\partial J}{\partial w^{[2]}}, \frac{\partial J}{\partial b^{[1]}}, \frac{\partial J}{\partial b^{[2]}}\right) = \begin{pmatrix} 2\cdot 1\cdot 1\cdot 3\cdot 1\cdot 1\cdot x \\ 2\cdot 1\cdot 1\cdot 2 \\ 2\cdot 1\cdot 1\cdot 3\cdot 1\cdot 1 \\ 2\cdot 1\cdot 1 \end{pmatrix} = \begin{pmatrix} 6x \\ 4 \\ 6 \\ 2 \end{pmatrix}$