1 A little bit of history: Linear Perceptron

Mark 1 perceptron (Frank Rosenblatt, 1957):

- An image recognition apparatus;
- 400 photo cells
- Weights are potentiometers;
- Weights are changed by electric motors.

The New York Times, 1958: > [...] the embryo of an electronic computer that the Navy expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.

1.1 Training the perceptron (no human guidance)

Training cycle (2000 “epochs”):

- holding an image in front of the digital camera (eg. triangle, circle, square, ...);
- observing which of the two lamps lit up (binary classes);
- checking if the lamp is correct (arbitrarily chosen);
- sending “reward” or “penalty” signal.
- human operator only performs mechanical actions.
2 Multi-layer neural networks - Inference

- Given a $n$-layer neural network and its parameters $\Theta^1, \ldots, \Theta^L$ oraz $\beta^1, \ldots, \beta^L$, we calculate for $l \in \{1, \ldots, L\}$: 
  \[
da^l = g^l \left( \Theta^l a^{l-1} + \beta^l \right) \]

- Parameters $\Theta^l$, weights on connection between neurons of layers $a^{l-1}$ and $a^l$, have size $\text{dim}(a^l) \times \text{dim}(a^{l-1})$.
- Bias vectors $\beta$ replace columns with “1” in feature matrix. The size of $\beta^l$ is equal to the size of the corresponding layer $\text{dim}(a^l)$.
- Function $g^l$ is the so called activation function:
- For $i = 0$ we assume $a^0 = x$ (features or input layer) and $g^0(x) = x$ (identity);
- In the case of classifiers, for the last layer $L$ often $g^L(x) = \text{softmax}(x)$;
- Other activation functions are often sigmoids (e.g. logistic function or hyperbolic tangens, tanh);
- In the case of regression networks, the last layer consists often of a single neuron.

2.1 Training multi-layer networks

- Parameters:
  \[
  \Theta = (\Theta^1, \Theta^2, \Theta^3, \beta^1, \beta^2, \beta^3) 
  \]

- Model:
  \[
  h_\Theta(x) = \tanh(\Theta^3 \tanh(\Theta^2 x + \beta^1) + \beta^2) + \beta^3 
  \]
* Cost function (MSE):

\[ J(\Theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\Theta(x^{(i)}) - y^{(i)})^2 \]

* How do we calculate the gradients?

\[ \nabla_{\Theta l} J(\Theta) = ? \quad \nabla_{\beta l} J(\Theta) = ? \quad l \in \{1, 2, 3\} \]

3 Backpropagation

- A hypothetical change \( \Delta z^l_j \) added to the \( j \)-th neuron in layer \( l \) propagates through the network and causes cost change:

\[ \frac{\partial J(\Theta)}{\partial z^l_j} \Delta z^l_j \]

- If \( \frac{\partial J(\Theta)}{\partial z^l_j} \) is large, \( \Delta z^l_j \) with an opposite sign can reduce the cost.
- If \( \frac{\partial J(\Theta)}{\partial z^l_j} \) is close to zero, the cost cannot be much improved.
- We define the error \( \delta^l_j \) of neuron \( j \) in layer \( l \):

\[ \delta^l_j \equiv \frac{\partial J(\Theta)}{\partial z^l_j} \quad \delta^l \equiv \nabla_{z^l} J(\Theta) \text{ (vectorized)} \]

3.1 The four fundamental equations of Backpropagation (proofs anyone?)

\[
\begin{align*}
\delta^L &= \nabla_{a^L} J(\Theta) \odot (g^L)'(z^L) \quad (BP1) \\
\delta^l &= ((\Theta^{l+1})^T \delta^{l+1}) \odot (g^l)'(z^l) \quad (BP2) \\
\nabla_{\beta^l} J(\Theta) &= \delta^l \quad (BP3) \\
\nabla_{\Theta^l} J(\Theta) &= a^{l-1} \odot \delta^l \quad (BP4)
\end{align*}
\]

3.2 The Backpropagation Algorithm

For one training example \((x, y)\):

1. Input: Set the activations of the input layers \( a^{0} = x \)
2. Forward step: for \( l = 1, \ldots, L \) calculate

\[ z^l = \Theta^{(l)} a^{l-1} + \beta^l \text{ and } a^l = g^l(z^l) \]

3. Output error \( \delta^L \): calculate vector

\[ \delta^L = \nabla_{a^L} J(\Theta) \odot (g^L)'(z^L) \]

4. Error backpropagation: for \( l = L - 1, L - 2, \ldots, 1 \) calculate

\[ \delta^l = ((\Theta^{l+1})^T \delta^{l+1}) \odot (g^l)'(z^l) \]

5. Gradients:

\[ \nabla_{\Theta^l} J(\Theta) = a^{l-1} \odot \delta^l \text{ and } \nabla_{\beta^l} J(\Theta) = \delta^l \]
3.3 SGD with Backpropagation

One iteration: * For all parameters $\Theta = (\Theta^1, \ldots, \Theta^L)$ create zero-valued helper matrices $\Delta = (\Delta^1, \ldots, \Delta^L)$ of the same size ($\beta$ omitted for simplicity). * For $m$ examples in the batch, $i = 1, \ldots, m$: * Perform backpropagation for example $(x^{(i)}, y^{(i)})$ and store the gradients $\nabla_{\Theta} J^{(i)}(\Theta)$ * $\Delta := \Delta + \frac{1}{m} \nabla_{\Theta} J^{(i)}(\Theta)$ * Update the weights: $\Theta := \Theta - \alpha \Delta$

3.4 What about more complicated networks?

- Backpropagation is usually formulated in the language of (Feedforward) Neural Networks (layers, weights, biases, activations, weighted inputs, ...)
- Today’s NNs contain more complicated operation, e.g. concatenation of bidirectional RNN states, ...
- But: what’s the derivation of the “concatenation” operation and where does that fit into the BP equations?

4 Reverse-mode Autodiff

4.1 Let’s calculate gradients for anything ... automatically!

$$f(x_1, x_2) = \sin(x_1) + x_1 x_2$$
4.2 An example computation graph

4.3 Forward propagations of values
4.4 The idea of reverse-mode auto-differentiation:

- Repeatedly substitute the derivative of the outer functions in the chain rule;
- Sub-expression follow the structure of the computation graph.

\[
\frac{\partial f}{\partial x} = \frac{\partial f}{\partial w_1} \frac{\partial w_1}{\partial x} = \left( \frac{\partial f}{\partial w_2} \frac{\partial w_2}{\partial w_1} \right) \frac{\partial w_1}{\partial x} = \left( \left( \frac{\partial f}{\partial w_3} \frac{\partial w_3}{\partial w_2} \right) \frac{\partial w_2}{\partial w_1} \right) \frac{\partial w_1}{\partial x} = \ldots
\]

- We calculate the adjoint:

\[ \bar{w} = \frac{\partial f}{\partial w} \]

4.5 Back propagation of adjoints

4.6 2-layer Neural Network

```cpp
auto x = input(shape={whatevs, 784});
auto y = input(shape={whatevs, 10});
```
auto w1 = param(shape={784, 100});
auto b1 = param(shape={1, 100});
auto l1 = tanh(dot(x, w1) + b1);

auto w2 = param(shape={100, 10});
auto b2 = param(shape={1, 10});
auto l2 = softmax(dot(l1, w2) + b2, axis=1);

auto graph = -mean(sum(y * log(l2), axis=1), axis=0);

x = Tensor({500, 784}, 1);
y = Tensor({500, 10}, 1);

graph.forward();
graph.backward();

auto dw = w.grad();
auto db = b.grad();

4.7 Unary node for Tanh operation in Marian

struct TanhNodeOp : public UnaryNodeOp {
  template <typename ...Args>
  TanhNodeOp(Args ...args) : UnaryNodeOp(args...) { }

  void forward() {
    Element(_1 = Tanh(_2),
           val_, a_->val());
  }

  void backward() {
    Element(_1 += _2 * (1 - Tanh(_3) * Tanh(_3)),
            a_->grad(), adj_, a_->val());
  }
};

4.8 Binary node for Division operation in Marian

struct DivNodeOp : public BroadcastingNodeOp {
  template <typename ...Args>
  DivNodeOp(Args ...args) : BroadcastingNodeOp(args...) { }

  void forward() {
    Element(_1 = _2 / _3,
            val_, a_->val(), b_->val());
  }

  void backward() {
    Element(_1 += _2 * 1.0f / _3,
            a_->grad(), adj_, b_->val());
    Element(_1 -= _2 * _3 / (_4 * _4),
            b_->grad(), adj_, a_->val(), b_->val());
  }
4.9 Complex Softmax node defined by other operators

```cpp
template <typename ...Args>
inline Expr softmax(Expr a, Args ...args) {
    Expr e = exp(a);
    return e / sum(e, args...);
}
```