Introduction to concepts in machine learning

- Cost functions
- Model complexity and overfitting
- Training set, validation set, test set
- Maximum-likelihood classification

Machine learning basics

- k-nearest neighbors
- Decision boundaries
- Classifiers based on linear models
- Softmax classifier
- Maximum-likelihood
- Support vector machines
- Hinge loss function

Gradient Descent

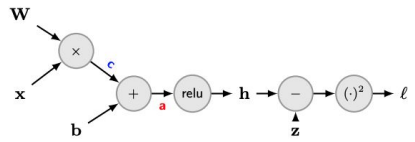
- Derivation and intuitions
- Hessian

Neural Networks

- Feedforward neural network architectures
- The importance of nonlinearity
- Example: XOR
- Activation functions
- Output units

Backpropagation: neural network layer

Here, $\mathbf{h} = \text{ReLU}(\mathbf{W}\mathbf{x} + \mathbf{b})$, with $\mathbf{h} \in \mathbb{R}^h$, $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{W} \in \mathbb{R}^{h \times m}$, and $\mathbf{b} \in \mathbb{R}^h$. While many output cost functions might be used, let's consider a simple squared-loss output $\ell = (\mathbf{h} - \mathbf{z})^2$ where $\mathbf{z}$ is some target value.



A few things to note:

- Note that $\nabla_{\mathbf{h}} \ell = 2(\mathbf{h} - \mathbf{z})$. We will start backpropagation at $\mathbf{h}$ rather than at ℓ.
- In the following backpropagation, we'll have need for elementwise multiplication. This is formally called a Hadamard product, and we will denote it via $\odot$. Concretely, the ith entry of $\mathbf{x} \odot \mathbf{y}$ is given by $x_i y_i$.

Applying the chain rule, we have:

$$\begin{aligned}
\frac{\partial \ell}{\partial \mathbf{a}} &= \mathbb{I}(\mathbf{a} > 0) \odot \frac{\partial \ell}{\partial \mathbf{h}} \\
\frac{\partial \ell}{\partial \mathbf{c}} &= \frac{\partial \ell}{\partial \mathbf{a}} \\
\frac{\partial \ell}{\partial \mathbf{x}} &= \frac{\partial \mathbf{c}}{\partial \mathbf{x}} \frac{\partial \ell}{\partial \mathbf{c}} \\
&= \mathbf{W}^T \frac{\partial \ell}{\partial \mathbf{c}}
\end{aligned}$$

Interpreting backpropagation as gradient "gates":

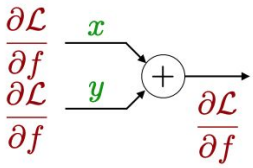
Add gate: distributes the gradient

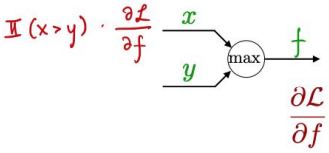
Interpreting backpropagation as gradient "gates":

Add gate: distributes the gradient
Mult gate: switches the gradient

$f = \max(x, y)$

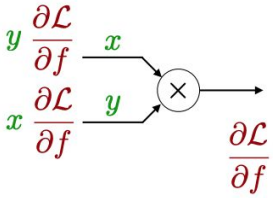
$\frac{\partial f}{\partial x} = \begin{cases} 1 & \text{if } x > y \\ 0 & \text{else} \end{cases} = \mathbb{I}(x > y)$





Interpreting backpropagation as gradient "gates":

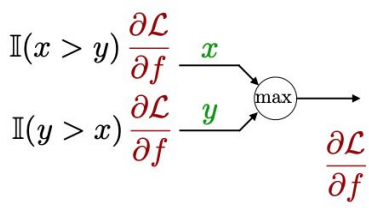
Add gate: distributes the gradient
Mult gate: switches the gradient



Interpreting backpropagation as gradient "gates":

Add gate: distributes the gradient
Mult gate: switches the gradient
Max gate: routes the gradient

if $x > y$



Training

- Weight initialization
  - Xavier initialization
- Pre-processing (Zero mean the data)

Batch normalization (cont.)

(Ioffe and Szegedy, 2015), introduced batch normalization.

- Normalize the unit activations:

$$\hat{x}_i = \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}}$$

with

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)} \qquad \sigma_i^2 = \frac{1}{m}\sum_{j=1}^{m}\left(x_i^{(j)} - \mu_i\right)^2$$

and ε small.

- Scale and shift the normalized activations:

$$y_i = \gamma_i \hat{x}_i + \beta_i$$

Importantly, the normalization and scale / shift operations are included in the computational graph of the neural network, so that they participate not only in forward propagation, but also in backpropagation.

Types of regularization

Regularizations may take on many different types of forms. The following list is not exhaustive, but includes regularizations one may consider.

- It may be appropriate to add a soft constraint on the parameter values in the objective function.
  - To account for prior knowledge (e.g., that the parameters have a bias).
  - To prefer simpler model classes that promote generalization.
  - To make an underdetermined problem determined. (e.g., least squares with indeterminate $\mathbf{X}^T\mathbf{X}$.)
- Dataset augmentation
- Ensemble methods (i.e., essentially combining the output of several models).
- Some training algorithms (e.g., stopping training early, dropout) can be seen as a type of regularization.

Batchnorm: Higher learning rates, robust to inits: Normalized activations => no exploding/vanishing activations due to variance in activations being too low or too high=> no exploding/vanishing gradients

An obstacle to standard training is that the distribution of the inputs to each layer changes as learning occurs in previous layers. As a result, the unit activations can be very variable. Another consideration is that when we do gradient descent, we're calculating how to update each parameter assuming the other layers don't change. But these layers may change drastically. Ultimately, these cause:

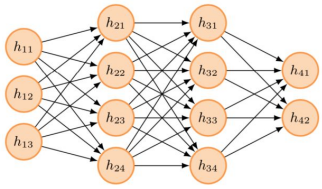
- Learning rates to be smaller (than if the distributions were not so variable).
- Networks to be more sensitive to initializations.
- Difficulties in training networks that saturate (where learning will no longer occur).

The idea of batch-normalization is to make the output of each layer have mean zero and standard deviation 1 statistics. Learning then becomes simpler because parameters in the lower layers do not change the statistics of the input to a given layer. This makes learning more straightforward.

- **L2 regularization**: encourages small weights ("weight decay")
- **L1 regularization**: encourages weights to go to 0 (makes model "sparser")
- **Dataset augmentation**: rotate, shift, add noise to dataset to effectively get more data
- **Multitask learning** (train model to perform multiple tasks) and **transfer learning** (build on top of a pre-trained network)
- **Dropout**
  - Regularizes: Each unit has a chance of being dropped to zero, so units must work well in many contexts
  - Approximates bagging (ensembling): each mask is like a different model

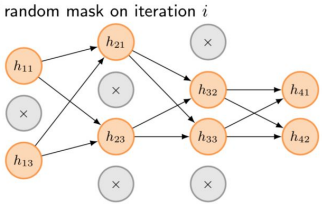
Dropout

Hidden layers of network to be trained



13 units

Application of random mask on iteration i



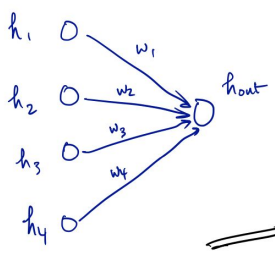
mask $\in \mathbb{R}^{13}$

2^N configs where N is the # of neurons.

Prof J.C. Kao, UCLA EC

Dropout

How about during test time? What configuration do you use?



$p = 0.5$

Training iter 1:

$M = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} \Rightarrow h_{out} = \text{relu}(w_1 h_1 + w_3 h_3)$

T.I. 2

$M = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \Rightarrow h_{out} = \text{relu}(w_2 h_2 + w_4 h_4)$

Test:

$h_{out} = \text{relu}\left((w_1 h_1 + w_2 h_2 + w_3 h_3 + w_4 h_4)p\right)$

Over many iters, the contribution of $w_i h_i$ to h_{out} is actually $p \cdot w_i \cdot h_i$

Prof J.C. Kao, UCLA ECE

Adam (cont.)

Initialize $\mathbf{v} = 0$ as the "first moment", and $\mathbf{a} = 0$ as the "second moment." Set β_1 and β_2 to be between 0 and 1. (Suggested defaults are $\beta_1 = 0.9$ and $\beta_2 = 0.999$.) Initialize ν to be sufficiently small. Initialize $t = 0$. Until stopping criterion is met:

- Compute gradient: $\mathbf{g}$
- Time update: $t \leftarrow t + 1$
- First moment update (momentum-like):

$$\mathbf{v} \leftarrow \beta_1 \mathbf{v} + (1 - \beta_1)\mathbf{g}$$

- Second moment update (gradient normalization):

$$\mathbf{a} \leftarrow \beta_2 \mathbf{a} + (1 - \beta_2)\mathbf{g} \odot \mathbf{g}$$

- Bias correction in moments:

$$\tilde{\mathbf{v}} = \frac{1}{1 - \beta_1^t}\mathbf{v}$$

$$\tilde{\mathbf{a}} = \frac{1}{1 - \beta_2^t}\mathbf{a}$$

- Gradient step:

$$\theta \leftarrow \theta - \frac{\varepsilon}{\sqrt{\tilde{\mathbf{a}}} + \nu} \odot \tilde{\mathbf{v}}$$

Momentum

In momentum, we maintain the running mean of the gradients, which then updates the parameters.

Initialize $\mathbf{v} = 0$. Set $\alpha \in [0, 1]$. Typical values are $\alpha = 0.9$ or 0.99. Then, until stopping criterion is met:

- Compute gradient: $\mathbf{g}$
- Update:

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \varepsilon \mathbf{g}$$

- Gradient step:

$$\theta \leftarrow \theta + \mathbf{v}$$

Adaptive gradient (Adagrad)

Adaptive gradient (Adagrad) is a form of stochastic gradient descent where the learning rate is decreased through division by the historical gradient norms. We will let the variable a denote a running sum of squares of gradient norms.

Initialize $\mathbf{a} = 0$. Set ν at a small value to avoid division by zero (e.g., $\nu = 1e - 7$). Then, until stopping criterion is met:

- Compute the gradient: $\mathbf{g}$
- Update:

$$\mathbf{a} \leftarrow \mathbf{a} + \mathbf{g} \odot \mathbf{g}$$

- Gradient step:

$$\theta \leftarrow \theta - \frac{\varepsilon}{\sqrt{\mathbf{a}} + \nu} \odot \mathbf{g}$$

A con of Adagrad is that eventually the step size will become sufficiently small when $\mathbf{a}$ grows too large such that no more meaningful learning occurs. This motivates the next optimizer, which shrinks $\mathbf{a}$ over time.

RMSProp

RMSProp augments Adagrad by making the gradient accumulator an exponentially weighted moving average.

Initialize $\mathbf{a} = 0$ and set ν to be sufficiently small. Set β to be between 0 and 1 (typically a value like 0.99). Then, until stopping criterion is met:

- Compute the gradient: $\mathbf{g}$
- Update:

$$\mathbf{a} \leftarrow \beta \mathbf{a} + (1 - \beta)\mathbf{g} \odot \mathbf{g}$$

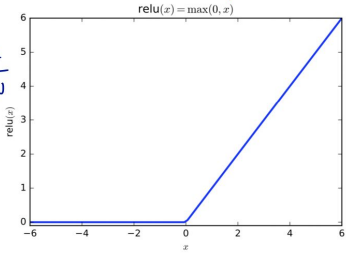
- Gradient step:

$$\theta \leftarrow \theta - \frac{\varepsilon}{\sqrt{\mathbf{a}} + \nu} \odot \mathbf{g}$$

Rectified linear unit, $\mathrm{ReLU}(x) = \max(0, x)$

```
f = lambda x: x * (x > 0)
```

np.maximum(0, x)



Its derivative is:

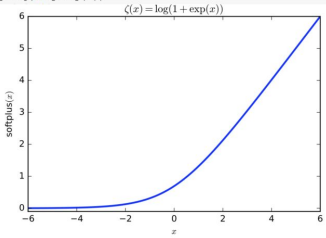
$$\frac{d\mathrm{ReLU}(x)}{dx} = \begin{cases} 1 & x > 0 \\ 0 & x \leqslant 0 \end{cases}$$

This function is not differentiable at $x = 0$. However, we can define its subgradient by setting the derivative to be between $[0, 1]$ at $x = 0$.

Softplus unit, $\zeta(x) = \log(1 + \exp(x))$

One may consider using the softplus function, $\zeta(x) = \log(1 + e^x)$, in place of $\mathrm{ReLU}(x)$. Intuitively, this ought to work well as it resembles $\mathrm{ReLU}(x)$ and is differentiable everywhere. However, empirically, it performs worse than $\mathrm{ReLU}(x)$.

```
f = lambda x: np.log(1+np.exp(x))
```



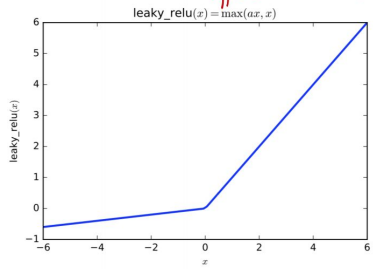
Its derivative is:

$$\frac{d\zeta(x)}{dx} = \sigma(x)$$

Leaky rectified linear unit, $f(x) = \max(\alpha x, x)$

```
f = lambda x: x * (x>0) + 0.1*x * (x<0)
```

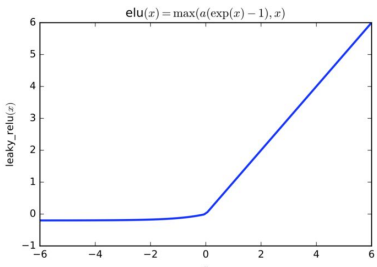
typical $\alpha = 0.01$



The leaky ReLU avoids the stopping of learning when $x < 0$. α may be treated as a selected hyperparameter, or it may be a parameter to be optimized in learning, in which case it is typically called the "PReLU" for parametrized rectified linear unit.

Exponential linear unit, $f(x) = \max(\alpha(\exp(x) - 1), x)$

```
f = lambda x: x * (x>0) + 0.2*(np.exp(x) - 1) * (x<0)
```



The exponential linear unit avoids the stopping of learning when $x < 0$. A con of this activation function is that it requires computation of the exponential, which is more expensive.

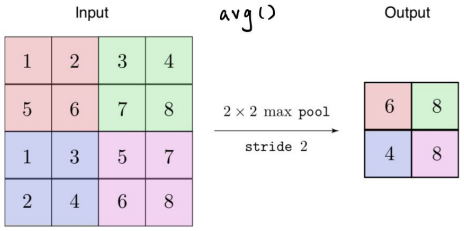
ConvNets

- Convolution operation
- Convolutional layer
  - Can handle variable size inputs, whereas FC cannot
  - Sparse interactions pros:
    - Reduce computational memory
    - Reduce computation time
  - Sparse interaction cons:
    - Shared parameters = less expressiveness
    - Each neuron has a small receptive field
- Pooling Layer

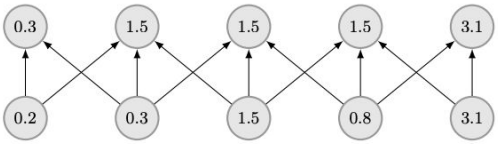
Pooling layer

CNNs also incorporate pooling layers, where an operation is applied to all elements within the filtering extent. This corresponds, effectively, to downsampling.

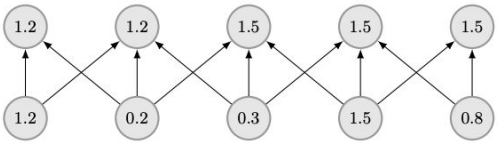
The pooling filter has width and height (w_p, h_p) and is applied with a given stride. It is most common to use the $\max()$ operation as the pooling operation.



Invariance example:



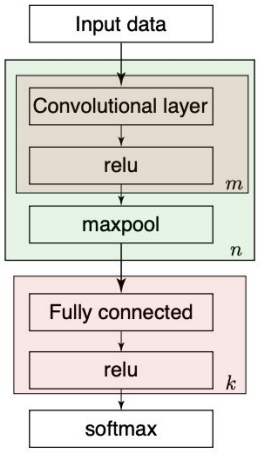
Inputs shifted by 1. Five inputs change but only three outputs change.



- Padding and stride
  - Convolutional dimensions: $\frac{w-w_f+2pad}{stride}+1$
  - Pooling dimensions: $\frac{w-w_f}{stride}+1$

Convolutional neural network architectures (cont.)

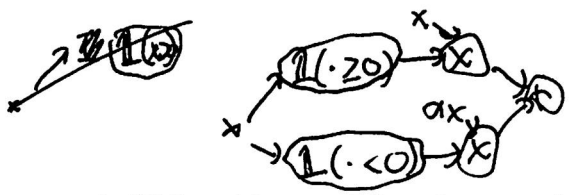
With this in mind, the following describes a fairly typical CNN architecture.



$W_1 x \in \mathbb{R}^n$

$W_2 W_1 x \in \mathbb{R}^n$

$W_3 W_2 W_1 x \in \mathbb{R}^m$



2. **Backpropagation.** Consider a 3 layer neural network (NN), with $\mathbf{x} \in \mathbb{R}^n$ as input and $\mathbf{y} \in \mathbb{R}^m$ as the target value. The NN is constructed as the following:

$$
\begin{aligned}
\mathcal{L} &= \frac{1}{2}||h_3(h_2(h_1(\mathbf{x}))) - \mathbf{y}||^2 \\
h_1(\mathbf{x}) &= \text{PReLU}(\mathbf{W}_1\mathbf{x}) \\
h_2(\mathbf{x}) &= \text{ELU}(\mathbf{W}_2\mathbf{x}) \\
h_3(\mathbf{x}) &= \mathbf{W}_3\mathbf{x}
\end{aligned}
$$

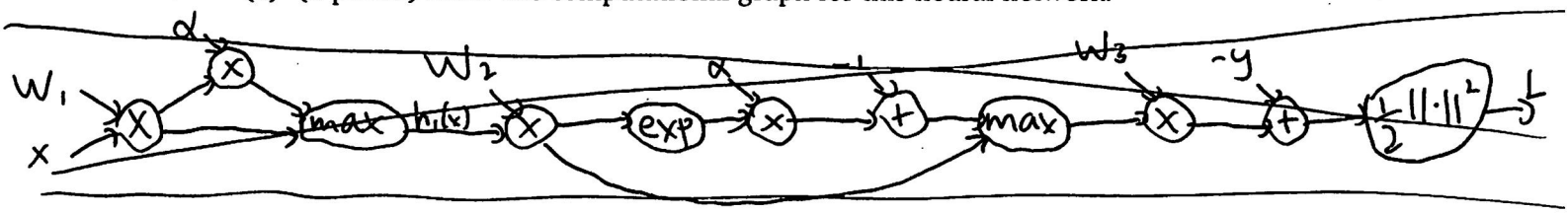
where

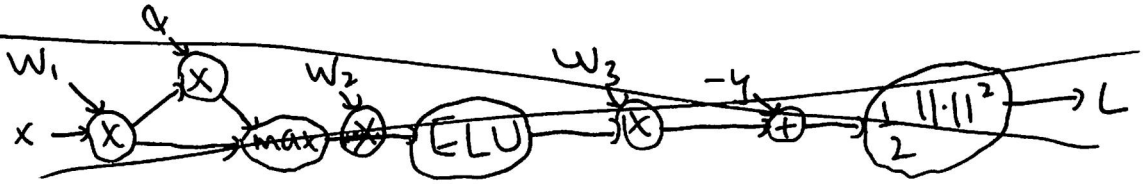
- $\mathbf{W}_1, \mathbf{W}_2 \in \mathbb{R}^{n \times n}$ and $\mathbf{W}_3 \in \mathbb{R}^{m \times n}$
- For the PReLU unit, $f(x) = \max(\alpha x, x)$.
- For the ELU unit, $f(x) = \max(\alpha e^x - 1, x)$.

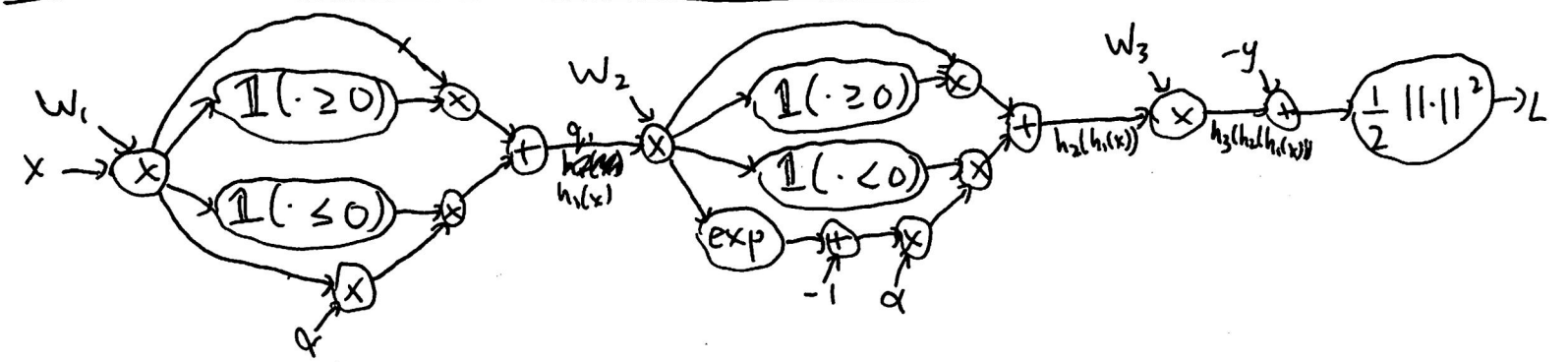
$$\text{ELU}(x) = \begin{cases} x, & x \geq 0 \\ \alpha(e^x - 1), & x < 0 \end{cases}$$

$$\text{PReLU}(x) = \begin{cases} x, & x \geq 0 \\ \alpha x, & x < 0 \end{cases}$$

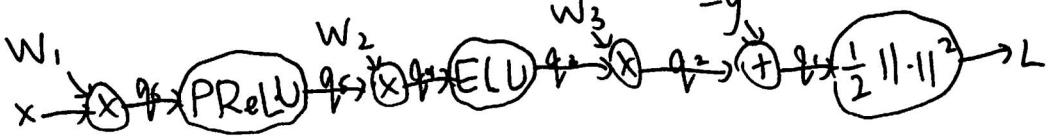
(a) (5 points) Draw the computational graph for this neural network.







More compactly:



W_3 $m\times n$ $\quad n\times 1$ $\quad m\times 1$ $\quad 1\times n$ $\quad f = W_3 x$ $\quad \frac{\partial L}{\partial x} = W^T \frac{\partial f}{\partial x}$ $\quad \frac{\partial f}{\partial x}: m\times n$

$W_3 x: m\times 1$ $\quad \frac{\partial L}{\partial W} = \frac{\partial f}{\partial W} x^T$

(b) (15 points) Calculate $\frac{\partial \mathcal{L}}{\partial \mathbf{W}_1}, \frac{\partial \mathcal{L}}{\partial \mathbf{W}_2}, \frac{\partial \mathcal{L}}{\partial \mathbf{W}_3}$ using backpropagation. You may define intermediate variables and write the gradients in terms of these intermediate variables.

Hint: For a ReLU function $f(x) = \max(0, x)$, we compute the gradient for $x < 0$ and $x > 0$ separately. For instance, $\frac{\partial f(x)}{\partial x} = \mathbb{1}\{x > 0\} \cdot 1 + \mathbb{1}\{x < 0\} \cdot 0$. Compute the gradient for PReLU and ELU with the same method. (Additional space provided on next page.)

Intermediate variables: $q_6 = W_1 x$; $q_5 = \text{PReLU}(q_6)$; $q_4 = W_2 q_5$; $q_3 = \text{ELU}(q_4)$;

$q_2 = W_3 q_3$; $q_1 = q_2 - y$; $L = \frac{1}{2}\|q_1\|^2$.

$$\frac{dL}{dq_1} = \frac{d}{dq_1}\left(\frac{1}{2}\|q_1\|^2\right) = \frac{1}{2}\cdot 2q_1 = q_1$$

$$\frac{dL}{dq_2} = \frac{dL}{dq_1}\frac{dq_1}{dq_2} = \frac{dL}{dq_1} = q_1$$

$$\frac{dL}{dW_3} = \frac{dL}{dq_2} q_3^T = q_1 q_3^T; \quad \frac{dL}{dq_3} = W_3^T \frac{dL}{dq_2} = W_3^T q_1$$

$$\frac{dq_3}{dq_4} = \frac{d\,\text{ELU}(q_4)}{dq_4} = \mathbb{1}\{q_4 > 0\}\cdot 1 + \mathbb{1}\{q_4 \le 0\}\,\alpha e^{q_4} = p \text{ (a new variable)}$$

So $$\frac{dL}{dq_4} = \frac{dL}{dq_3}\left(\mathbb{1}\{q_4 > 0\} + \mathbb{1}\{q_4 < 0\}\,\alpha e^{q_4}\right) = \frac{dL}{dq_3} p = W_3^T q_1 p$$

$$\frac{dL}{dW_2} = \frac{dL}{dq_4} q_5^T = W_3^T q_1 p q_5^T. \quad \frac{dL}{dq_5} = W_2^T \frac{dL}{dq_4} = W_2^T W_3^T q_1 p$$

$$\frac{dq_5}{dq_6} = \frac{d\,\text{PReLU}(q_6)}{dq_6} = \mathbb{1}\{q_6 > 0\}\cdot 1 + \mathbb{1}\{q_6 < 0\}\,\alpha = r \text{ (a new variable)}$$

$$\frac{dL}{dq_6} = \frac{dL}{dq_5} = W_2^T W_3^T q_1 p r$$

$$\frac{dL}{dW_1} = \frac{dL}{dq_6} x^T = W_2^T W_3^T q_1 p r x^T$$