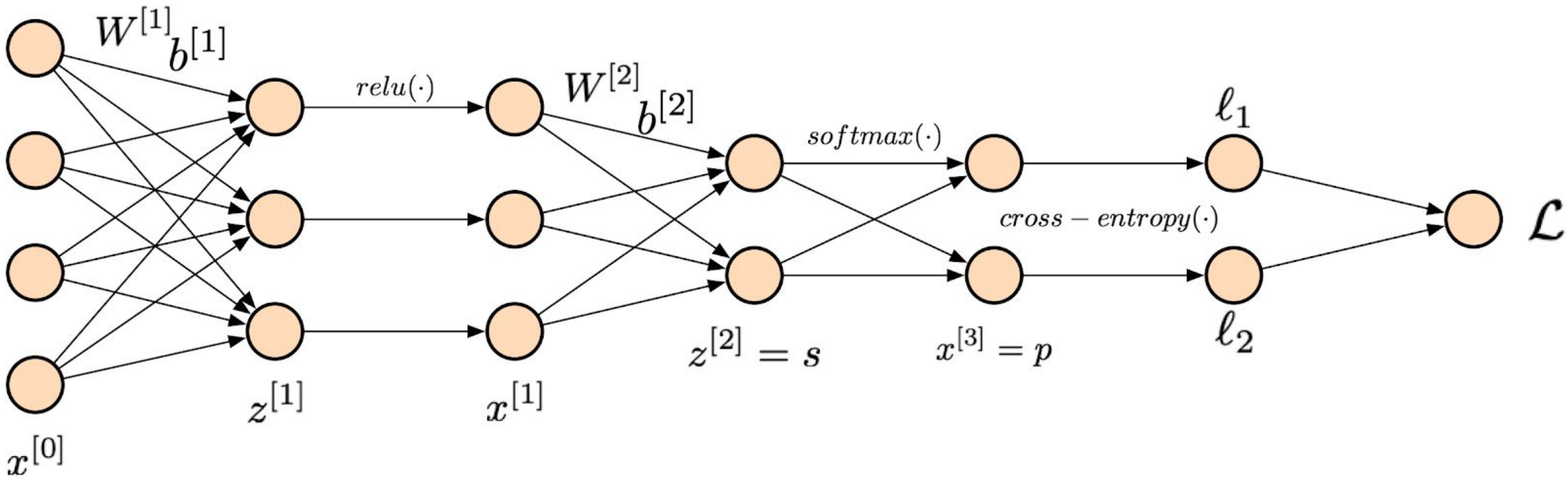
Lab3 Solution: Neural Networks

Notes on the vectorized form of the gradients (needed for function "*loss*" in the python class "*TwoLayerNetwork*")



Non Vectorized Gradients:

$$\frac{\partial \mathcal{L}}{\partial p_i} = \frac{\partial}{\partial p_i}\left\{\ell_i\right\} = \frac{\partial}{\partial p_i}\left\{y_i \log(p_i)\right\} = -\frac{y_i}{p_i}$$

$$\frac{\partial \mathcal{L}}{\partial s_k} = \sum_i \frac{\partial \mathcal{L}}{\partial p_i}\frac{\partial p_i}{\partial s_k} = -\sum_i \frac{y_i}{p_i}\frac{\partial}{\partial s_k}\left\{\frac{e^{s_i}}{\sum_l e^{s_l}}\right\} = -\sum_i \frac{y_i}{p_i}\left[p_i\left(\delta_{i=k}(1-p_k)\right) - \delta_{i\neq k}p_k\right] = -\sum_i y_i(\delta_{i=k} - p_k) = p_k - y_k$$

$$\frac{\partial \mathcal{L}}{\partial w_{pk}^{[2]}} = \frac{\partial \mathcal{L}}{\partial s_k}\frac{\partial s_k}{\partial w_{pk}^{[2]}} + 2\lambda w_{pk}^{[2]} = (p_k - y_k)\frac{\partial}{\partial w_{pk}^{[2]}}\left\{\sum_r x_r^{[1]} w_{rk}^{[2]} + b_k^{[2]}\right\} + 2\lambda w_{pk}^{[2]} = (p_k - y_k)x_p^{[1]} + 2\lambda w_{pk}^{[2]}$$

$$\frac{\partial \mathcal{L}}{\partial b_k^{[2]}} = \frac{\partial \mathcal{L}}{\partial s_k}\frac{\partial s_k}{\partial b_k^{[2]}} + 2\lambda b_k^{[2]} = (p_k - y_k)\frac{\partial}{\partial b_k^{[2]}}\left\{\sum_r x_r^{[1]} w_{rk}^{[2]} + b_k^{[2]}\right\} + 2\lambda b_k^{[2]} = (p_k - y_k) + 2\lambda b_k^{[2]}$$

$$\frac{\partial \mathcal{L}}{\partial x_t^{[1]}} = \sum_k \frac{\partial \mathcal{L}}{\partial s_k}\frac{\partial s_k}{\partial x_t^{[1]}} = \sum_k (p_k - y_k) w_{tk}^{[2]}$$

$$\frac{\partial \mathcal{L}}{\partial z_t^{[1]}} = \frac{\partial \mathcal{L}}{\partial x_t^{[1]}}\frac{\partial x_t^{[1]}}{\partial z_t^{[1]}} = \sum_k (p_k - y_k) w_{tk}^{[2]} \mathbb{1}\{z_t^{[1]} \geq 0\}$$

$$\frac{\partial \mathcal{L}}{\partial w_{mt}^{[1]}} = \frac{\partial \mathcal{L}}{\partial z_t^{[1]}}\frac{\partial z_t^{[1]}}{\partial w_{mt}^{[1]}} + 2\lambda w_{mt}^{[1]} = \sum_k (p_k - y_k) w_{tk}^{[2]} \mathbb{1}\{z_t^{[1]} \geq 0\} x_t^{[0]} + 2\lambda w_{mt}^{[1]}$$

$$\frac{\partial \mathcal{L}}{\partial b_t^{[1]}} = \frac{\partial \mathcal{L}}{\partial z_t^{[1]}}\frac{\partial z_t^{[1]}}{\partial b_t^{[1]}} + 2\lambda b_t^{[1]} = \sum_k (p_k - y_k) w_{tk}^{[2]} \mathbb{1}\{z_t^{[1]} \geq 0\} + 2\lambda b_t^{[1]}$$

Vectorized Gradients:

$$\frac{\partial \mathcal{L}}{\partial W^{[2]}} = \frac{1}{N}\left(x^{[1]}\right)^T \cdot (p-y) + 2\lambda W^{[2]}$$

$$\frac{\partial \mathcal{L}}{\partial b^{[2]}} = \frac{1}{N}(p-y) \cdot \mathbf{1} + 2\lambda b^{[2]}$$

$$\frac{\partial \mathcal{L}}{\partial W^{[1]}} = \frac{1}{N}\left(x^{[0]}\right)^T \cdot \left[(p-y) \cdot (W^{[2]})^T \odot \mathbf{1}\{z^{[1]} \geq 0\}\right] + 2\lambda W^{[1]}$$

$$\frac{\partial \mathcal{L}}{\partial b^{[1]}} = \frac{1}{N}\left[(p-y) \cdot (W^{[2]})^T \odot \mathbf{1}\{z^{[1]} \geq 0\}\right] \cdot \mathbf{1} + 2\lambda b^{[1]}$$

```python
def loss(self, X, y=None, reg=0.0):

        # Unpack variables from the params dictionary
        W1, b1 = self.params['W1'], self.params['b1']
        W2, b2 = self.params['W2'], self.params['b2']
        N, D = X.shape

        # Compute the forward pass
        z1 = X.dot(W1) + b1
        x1 = np.maximum(z1, 0)
        scores = x1.dot(W2) + b2
        probs = np.exp(scores) / np.exp(scores).sum(axis=-1, keepdims=True)

        # If the targets are not given then jump out, we're done
        if y is None:
            return scores

        # Compute the loss
        y_oh = np.array([1.0 * (y_i == np.arange(probs.shape[-1])) for y_i in y], dtype=np.float32)
        loss = - (y_oh * np.log(probs)).sum(axis=-1).mean() + reg * (W1 ** 2).sum() + reg * (W2 ** 2).sum()

        # Backward pass: compute gradients
        grads = {}
        grads['W2'] = np.dot(x1.T, probs - y_oh) / N + 2 * reg * W2
        grads['b2'] = (probs - y_oh).sum(axis=0) / N + 2 * reg * b2
        grads['W1'] = np.dot(X.T, np.dot(probs - y_oh, W2.T) * 1.0 * (z1 >= 0)) / N + 2 * reg * W1
```

```
        grads['b1'] = (np.dot(probs - y_oh, W2.T) * 1.0 * (z1 >= 0)).sum(axis=0) / N + 2 * reg * b1

        return loss, grads
```

```
def train(self, X, y, X_val, y_val,
             learning_rate=1e-3, learning_rate_decay=0.95,
             reg=5e-6, num_iters=100,
             batch_size=200, verbose=False):

        num_train = X.shape[0]
        iterations_per_epoch = max(num_train / batch_size, 1)

        # Use SGD to optimize the parameters in self.model
        loss_history = []
        train_acc_history = []
        val_acc_history = []

        for it in range(num_iters):
            idx_batch = np.random.randint(num_train, size=batch_size)
            X_batch, y_batch = X[idx_batch], y[idx_batch]


            # Compute loss and gradients using the current minibatch
            loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
```

```python
      loss_history.append(loss)

      self.params['W1'] = self.params['W1'] - learning_rate * grads['W1']
      self.params['b1'] = self.params['b1'] - learning_rate * grads['b1']
      self.params['W2'] = self.params['W2'] - learning_rate * grads['W2']
      self.params['b2'] = self.params['b2'] - learning_rate * grads['b2']

      if verbose and it % 100 == 0:
        print('iteration %d / %d: loss %f' % (it, num_iters, loss))

      # Every epoch, check train and val accuracy and decay learning rate.
      if it % iterations_per_epoch == 0:
        # Check accuracy
        train_acc = (self.predict(X_batch) == y_batch).mean()
        val_acc = (self.predict(X_val) == y_val).mean()
        train_acc_history.append(train_acc)
        val_acc_history.append(val_acc)

        # Decay learning rate
        learning_rate *= learning_rate_decay

    return {
      'loss_history': loss_history,
      'train_acc_history': train_acc_history,
      'val_acc_history': val_acc_history,
    }
```

```python
def predict(self, X):
        z1 = X.dot(self.params['W1']) + self.params['b1']
        x1 = np.maximum(z1, 0)
        scores = x1.dot(self.params['W2']) + self.params['b2']
        probs = np.exp(scores) / np.exp(scores).sum(axis=-1, keepdims=True)
        y_pred = probs.argmax(axis=-1)
        return y_pred
```

```python
# Good network
input_size = 32 * 32 * 3
hidden_size = 256
num_classes = 10
best_net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
stats = best_net.train(X_train, y_train, X_val, y_val,
            num_iters=5000, batch_size=128,
            learning_rate=5e-4, learning_rate_decay=0.95,
            reg=0.15, verbose=True)
```

```python
# Predict on the validation set
val_acc = (net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```