Problem 1

1. In one-dimensional case, the cost C is some function of a single variable v.

$$\Delta C = \frac{dC}{dv} * \Delta v$$

When the derivative is positive, choose Δv to be negative (i.e., decrease v). When the derivative is negative, choose Δv to be positive (i.e., increase v). This way, it is guaranteed that ΔC is always negative, in other words, the cost C always decreases.

2. Advantage: More time and space efficient per update, since computing and storing gradients of one sample take much less time and space than with the entire dataset. This computational advantage allows for more iterations of SGD (e.g., more epochs).

Disadvantage: A mini-batch of size 1 captures much less information than the entire dataset, thus the gradients are less precise (i.e., noisier). When weights and biases are updated, they move less directly towards the optimal solution, thus are more likely to deviate from what they ought to be.

In practice, a mini-batch size somewhere between 1 and n is likely both to take advantage of the computational efficiency of SGD and to update the parameters in a more accurate manner.

3. prob1.py, mnist_loader.py, network.py

```
(venv) (base) Fans-MacBook-Pro:PS2 fanfeng$ python3 prob1.py
Epoch 0: 6714 / 10000
Epoch 1: 7372 / 10000
Epoch 2: 7622 / 10000
Epoch 3: 7660 / 10000
```

Epoch 4: 7661 / 10000
Epoch 5: 8193 / 10000
Epoch 6: 8330 / 10000
Epoch 7: 8357 / 10000
Epoch 8: 8374 / 10000
Epoch 9: 8371 / 10000
Epoch 10: 8361 / 10000
Epoch 11: 8338 / 10000
Epoch 12: 8353 / 10000
Epoch 13: 8377 / 10000
Epoch 14: 8352 / 10000
Epoch 15: 8384 / 10000
Epoch 16: 8384 / 10000
Epoch 17: 8370 / 10000
Epoch 18: 8391 / 10000
Epoch 19: 8381 / 10000
Epoch 20: 8384 / 10000
Epoch 21: 8392 / 10000
Epoch 22: 8359 / 10000
Epoch 23: 8384 / 10000
Epoch 24: 8375 / 10000
Epoch 25: 8391 / 10000
Epoch 26: 8373 / 10000
Epoch 27: 8378 / 10000
Epoch 28: 8342 / 10000
Epoch 29: 8385 / 10000

Accuracy is 83.85%, although different runs yield different results.

Problem 2

1. The Hadamard product ⊙ denotes the elementwise product of two vectors.

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

$$\delta_j^L = (\nabla_a C)_j * \sigma'\left(z_j^L\right) = \frac{\partial C}{\partial a_j^L} * \sigma'\left(z_j^L\right)$$

Now, let $\sum'(z^L)$ be a diagonal matrix where the j-th element along the diagonal is $\sigma'\left(z_j^L\right)$. Then, by **matrix multiplication**, $\sum'(z^L)\,\nabla_a C$ has the dimension as δ^L, and the j-th element equals $\sigma'\left(z_j^L\right) * \frac{\partial C}{\partial a_j^L}$.

Therefore, $\delta^L = \sum'(z^L)\,\nabla_a C$.

2. Similar to problem 2.1,

$$\delta^l = ((w^{l+1})^T\delta^{l+1}) \odot \sigma'(z^l)$$
$$\delta_j^l = ((w^{l+1})^T\delta^{l+1})_j * \sigma'\left(z_j^l\right)$$

Now, let $\sum'(z^l)$ be a diagonal matrix where the j-th element along the diagonal is $\sigma'\left(z_j^l\right)$. Then, by matrix multiplication, $\sum'(z^l)(w^{l+1})^T\delta^{l+1}$ has the dimension as δ^l, and the j-th element equals $\sigma'\left(z_j^L\right) * ((w^{l+1})^T\delta^{l+1})_j$.

Therefore, $\delta^l = \sum'(z^l)(w^{l+1})^T\delta^{l+1}$.

3. Applying results from problem 2.2 and 2.1,

$$\begin{aligned}
\delta^l &= \sum'(z^l)(w^{l+1})^T\delta^{l+1} \\
&= \sum'(z^l)(w^{l+1})^T\sum'(z^{l+1})(w^{l+2})^T\delta^{l+2} \\
&= \cdots \\
&= \sum'(z^l)(w^{l+1})^T\sum'(z^{l+1})(w^{l+2})^T \ldots \sum'(z^{L-1})(w^L)^T\delta^L \\
&= \sum'(z^l)(w^{l+1})^T\sum'(z^{l+1})(w^{l+2})^T \ldots \sum'(z^{L-1})(w^L)^T\sum'(z^L)\,\nabla_a C
\end{aligned}$$

4. When $\sigma(z) = z$, $\sigma'(z) = 1$, thus $\sum'(z) = I$. Therefore,

$$\delta^L = \sum'(z^L)\,\nabla_a C = \nabla_a C \qquad (BP1)$$
$$\delta^l = \sum'(z^l)(w^{l+1})^T\delta^{l+1} = (w^{l+1})^T\delta^{l+1} \qquad (BP2)$$

Note that one can chain $BP2$ in a similar way as in problem 2.3.

Finally, $BP3$ and $BP4$ remain the same.

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \qquad (BP3)$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \qquad (BP4)$$

Problem 3

1. When $y = 0$, $-\big(y\ ln(a) + (1-y)\ ln(1-a)\big) = -ln(1-a)$.

When $y = 1$, $-\big(y\ ln(a) + (1-y)\ ln(1-a)\big) = -ln(a)$.

In contrast, when $y = 0$, $-\big(a\ ln(y) + (1-a)\ ln(1-y)\big) = -a\ ln(0)$, which is undefined.

When $y = 1$, $-\big(a\ ln(y) + (1-a)\ ln(1-y)\big) = -(1-a)\ ln(0)$, which is undefined.

The problem with the second expression doesn't afflict the first expression.

2. The cross-entropy cost is,

$$C = -\frac{1}{n}\sum_x (y\ ln(a) + (1-y)ln(1-a))$$

Take the derivative of C with respect to a,

$$\frac{\partial C}{\partial a} = -\frac{1}{n}\sum_x \left(\frac{y}{a} - \frac{1-y}{1-a}\right)$$

When $a = \sigma(z) = y$, $\frac{\partial C}{\partial a} = 0$, thus the cross-entropy cost is minimized.

When $a = \sigma(z) = y$,

$$C = -\frac{1}{n}\sum_x \big(y\, ln(a) + (1-y) ln(1-a)\big)$$
$$= -\frac{1}{n}\sum_x (y\, ln(y) + (1-y) ln(1-y))$$

3.

$a_1^1 = \sigma(0.15 * 0.05 + 0.20 * 0.10 + 0.35) = \sigma(0.3775) = 0.5933$
$a_2^1 = \sigma(0.25 * 0.05 + 0.30 * 0.10 + 0.35) = \sigma(0.3925) = 0.5969$
$a_1^2 = \sigma(0.40 * 0.5933 + 0.45 * 0.5969 + 0.60)$
$= \sigma(1.1059) = 0.7514$
$a_2^2 = \sigma(0.50 * 0.5933 + 0.55 * 0.5969 + 0.60)$
$= \sigma(1.2249) = 0.7729$

$$C = -\frac{1}{2}(target_1 * ln(a_1^2) + (1 - target_1) * ln(1 - a_1^2))$$
$$-\frac{1}{2}(target_2 * ln(a_2^2) + (1 - target_2) * ln(1 - a_2^2))$$
$$C = -\frac{1}{2}(0.01 * ln(0.7514) + 0.99 * ln(0.2486))$$
$$-\frac{1}{2}(0.99 * ln(0.7729) + 0.01 * ln(0.2271))$$

$= 0.8254$

$$\delta^2 = \begin{pmatrix} \frac{\partial C}{\partial a_1^2} \\ \frac{\partial C}{\partial a_2^2} \end{pmatrix} \odot \begin{pmatrix} \frac{\partial a_1^2}{\partial z_1^2} \\ \frac{\partial a_2^2}{\partial z_2^2} \end{pmatrix}$$

$$= -\frac{1}{2}\begin{pmatrix} \frac{target_1}{a_1^2} - \frac{1 - target_1}{1 - a_1^2} \\ \frac{target_2}{a_2^2} - \frac{1 - target_2}{1 - a_2^2} \end{pmatrix} \odot \begin{pmatrix} a_1^2(1 - a_1^2) \\ a_2^2(1 - a_2^2) \end{pmatrix}$$

$$= -\frac{1}{2}\begin{pmatrix} \frac{0.01}{0.7514} - \frac{0.99}{0.2486} \\ \frac{0.99}{0.7729} - \frac{0.01}{0.2271} \end{pmatrix} \odot \begin{pmatrix} 0.7514 * 0.2486 \\ 0.7729 * 0.2271 \end{pmatrix}$$

$$= \begin{pmatrix} 0.3707 \\ -0.1086 \end{pmatrix}$$

The derivatives of the cost with respect to weights of the second layer are,

$$(0.3707 * a_1^1, 0.3707 * a_2^1, -0.1086 * a_1^1, -0.1086 * a_2^1)$$
$$= (0.2199, 0.2213, -0.0644, -0.0648)$$

The derivative of the cost with respect to biases of the second layer is (0.3707, -0.1086).

$$\delta^1 = ((w^2)^T \delta^2) \odot \sigma'(z^1)$$
$$= \begin{pmatrix} 0.40 & 0.50 \\ 0.45 & 0.55 \end{pmatrix} \begin{pmatrix} 0.3707 \\ -0.1086 \end{pmatrix} \odot \begin{pmatrix} 0.5933 * 0.4067 \\ 0.5969 * 0.4031 \end{pmatrix}$$
$$= \begin{pmatrix} 0.0489 \\ 0.0545 \end{pmatrix}$$

The derivative of the cost with respect to weights of the first layer are,
$$(0.0489 * 0.05, 0.0489 * 0.10, 0.0545 * 0.05, 0.0545 * 0.10)$$
$$= (0.0024, 0.0049, 0.0027, 0.0055)$$

The derivative of the cost with respect to biases of the first layer are, (0.0489, 0.0545).

$$\begin{aligned}\delta^0 &= ((w^1)^T\delta^1) \odot \sigma'(z^0) \\ &= \begin{pmatrix}0.15 & 0.25\\ 0.20 & 0.30\end{pmatrix}\begin{pmatrix}0.0489\\ 0.0545\end{pmatrix} \odot \begin{pmatrix}0.05\\ 0.10\end{pmatrix} \\ &= \begin{pmatrix}0.0010\\ 0.0026\end{pmatrix}\end{aligned}$$

Problem 4

1. See **prob4_1.py** and **ff242.ipynb**.

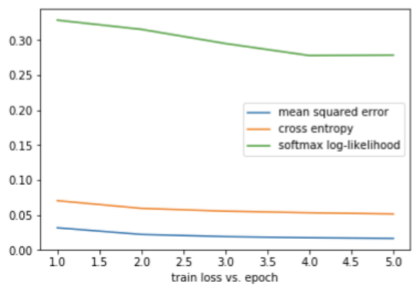
From the plots, it seems that mean squared error loss and cross entropy loss converge faster than softmax log-likelihood loss.

Besides, with 5 epochs, softmax log-likelihood loss and cross entropy loss achieve higher test accuracy than mean squared error loss.

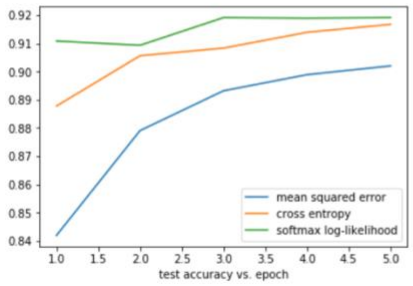
```
In [9]: import matplotlib.pyplot as plt
```

```
In [10]: x = [1, 2, 3, 4, 5]
train_loss_mse = [0.03163948, 0.022160674, 0.019042123, 0.017418016, 0.016385227]
test_accuracy_mse = [0.842, 0.8791, 0.8932, 0.8989, 0.902]

train_loss_ce = [0.07045822, 0.05945919, 0.05542801, 0.053181525, 0.051429845]
test_accuracy_ce = [0.8878, 0.9056, 0.9083, 0.9139, 0.9167]

train_loss_sll = [0.32885182, 0.31568593, 0.2954195, 0.2782145, 0.27890232]
test_accuracy_sll = [0.9108, 0.9093, 0.9191, 0.9189, 0.9191]
```

```
In [11]: plt.xlabel('train loss vs. epoch')
mse, = plt.plot(x, train_loss_mse, label='mean squared error')
ce, = plt.plot(x, train_loss_ce, label='cross entropy')
sll, = plt.plot(x, train_loss_sll, label='softmax log-likelihood')
plt.legend(handles=[mse, ce, sll])
plt.show()
```



```
In [12]: plt.xlabel('test accuracy vs. epoch')
         mse, = plt.plot(x, test_accuracy_mse, label='mean squared error')
         ce, = plt.plot(x, test_accuracy_ce, label='cross entropy')
         sll, = plt.plot(x, test_accuracy_sll, label='softmax log-likelihood')
         plt.legend(handles=[mse, ce, sll])
         plt.show()
```




```
In [ ]:
```

2. Note that softmax cross-entropy loss is used for this problem.

λ	L1 test accuracy	L2 test accuracy
0 (no regularization)	**0.9177**	**0.9177**
0.0001	0.9180	0.9167
0.0005	0.9101	0.9182
0.001	0.8909	0.9186
0.005	0.8549	0.9129
0.01	0.1135	0.9123
0.1	0.1135	0.8175

$p(keep)$	test accuracy
1.0 (no dropout)	**0.9177**
0.2	0.4825
0.4	0.7582
0.5	0.8225
0.6	0.8545
0.8	0.8945
0.9	0.9068

The final results are sensitive to the parameters.

Neural networks with regularizations learn nothing from the training data when λ becomes too large. L2 regularization is more tolerant on large λ then L1 regularization.

Dropout doesn't improve test accuracy because the neural networks don't overfit the data.