Lecture Plan:

- Recap of the backpropagation algorithm, discussion & extension
- Improving the way neural networks learn

3.1 Cross-entropy loss function

- learning slowdown: comparing Quadratic loss and cross-entropy loss
- How to derive the cost function

3.2 Overfitting and regularization

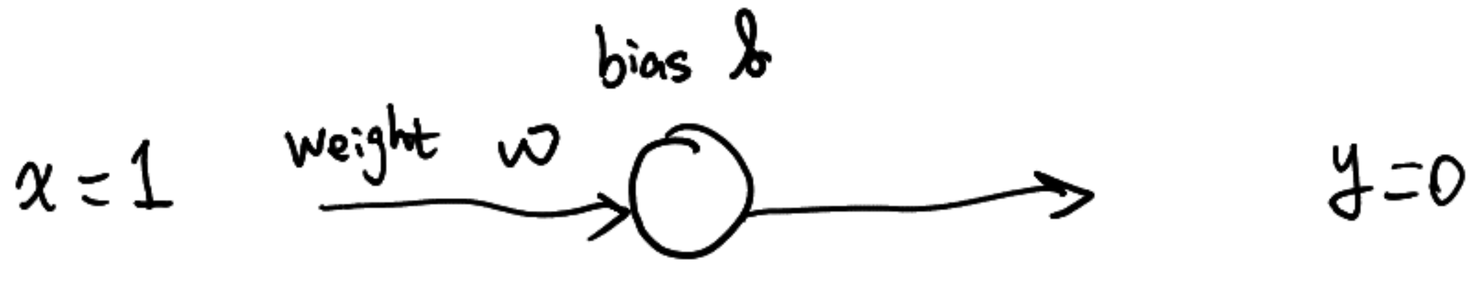
- An example of overfitting, and how to detect overfitting
- Regularization techniques
  - L_2 regularization
  - L_1 regularization
  - Dropout
  - Data Augmentation

3.3 Weight initialization

3.4 Variations on stochastic gradient descent

3.1. Cross-entropy Loss function

* Example: Comparing the quadratic loss and the cross-entropy loss for learning a sigmoid neuron.



IN — Sigmoid neuron — OUT

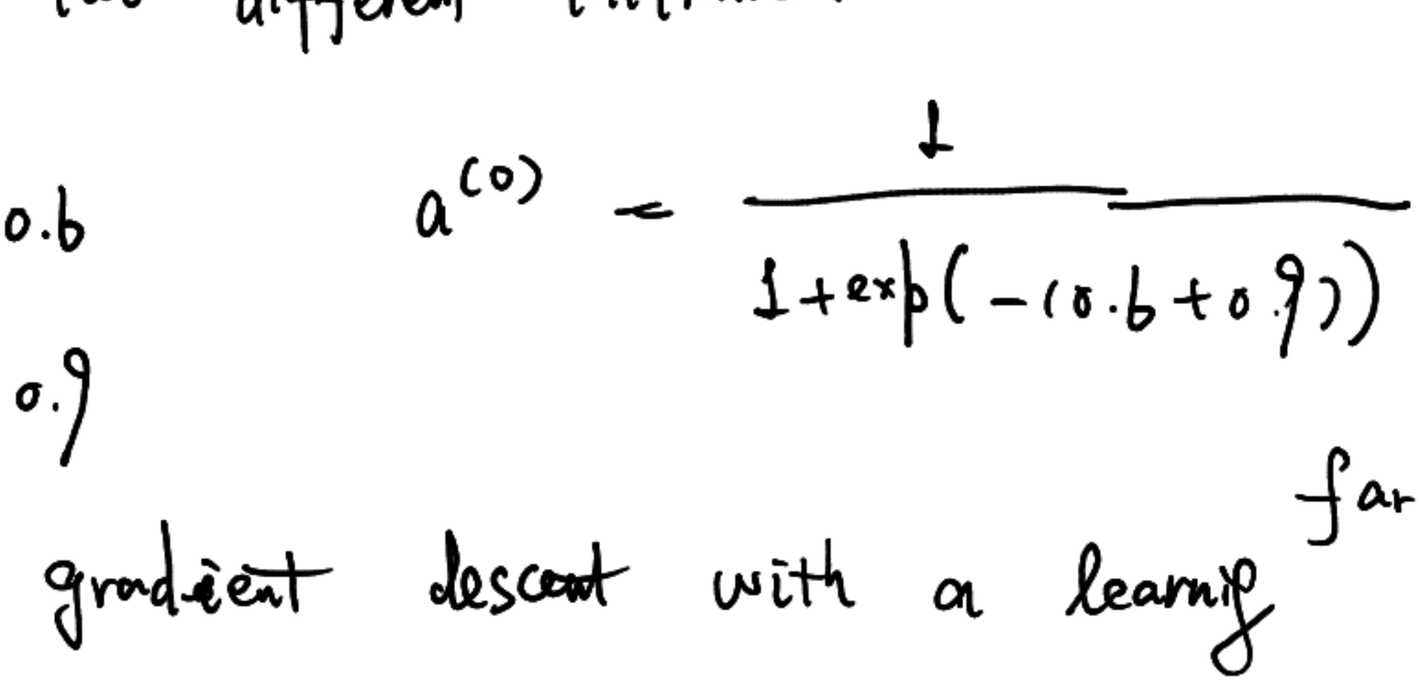
$$\sigma(z) = \frac{1}{1+\exp(-z)}$$

Quadratic loss. $C(w,b) = \frac{(a-y)^2}{2}$, where a is the output of the sigmoid neuron.

Let's compare two different initializations.

Init 1: $w^{(0)} = 0.6$, $b^{(0)} = 0.9$ $\quad a^{(0)} = \frac{1}{1+\exp(-(0.6+0.9))} \approx 0.8$.

If we use gradient descent with a learning rate $\eta = 0.15$, we get a result that looks like the following. (far from $y=0$)

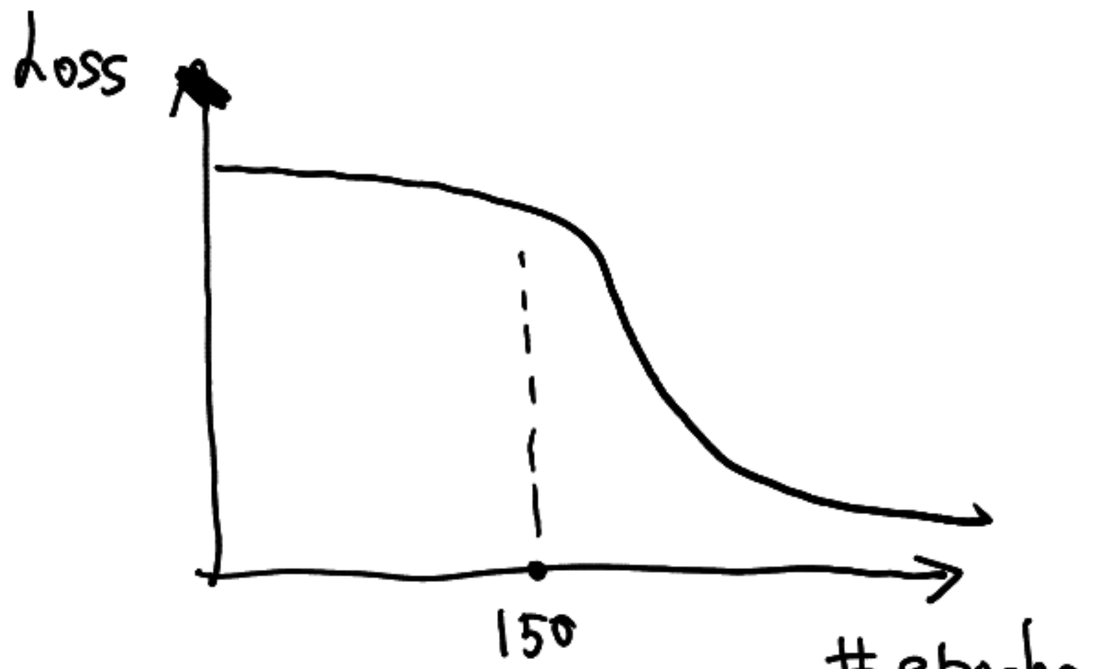


Init 2. $w^{(0)} = 2$

$b^{(0)} = 2$

$$a^{(0)} = \frac{1}{1+\exp(-(2.0+2.0))} \approx 0.98$$

very far from $y=0$!



Ex. Replicate this example in PyTorch

This is undesirable since when the error between a and y is large in the beginning, we expect the learning process to be faster ?

To understand this example a bit more, let's calculate the gradients of the loss.

$$L(w,b) = \frac{(a-y)^2}{2}, \qquad a = b(z) = b(w\cdot x + b)$$

$$\frac{\partial L}{\partial w} = (a-y)\cdot\frac{\partial a}{\partial w} = (a-y)\cdot b'(z)\cdot w,$$

$$\frac{\partial L}{\partial b} = (a-y)\cdot\frac{\partial a}{\partial b} = (a-y)\cdot b'(z).$$

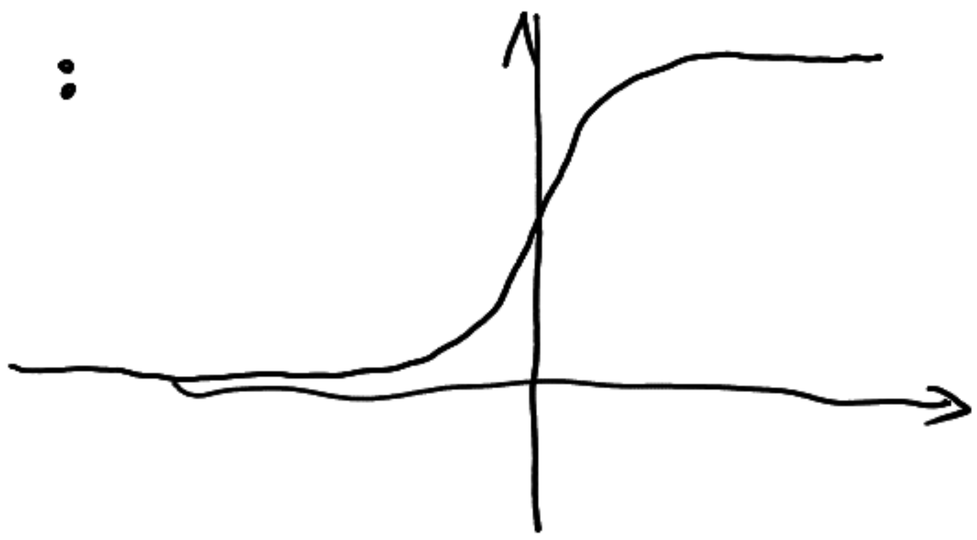
There are two terms that determine the magnitude of the gradients above :

(i) $a-y$: the error between the prediction and

and the true output.

(ii) $\sigma'(z)$: the gradient of the activation function.

Sigmoid function :

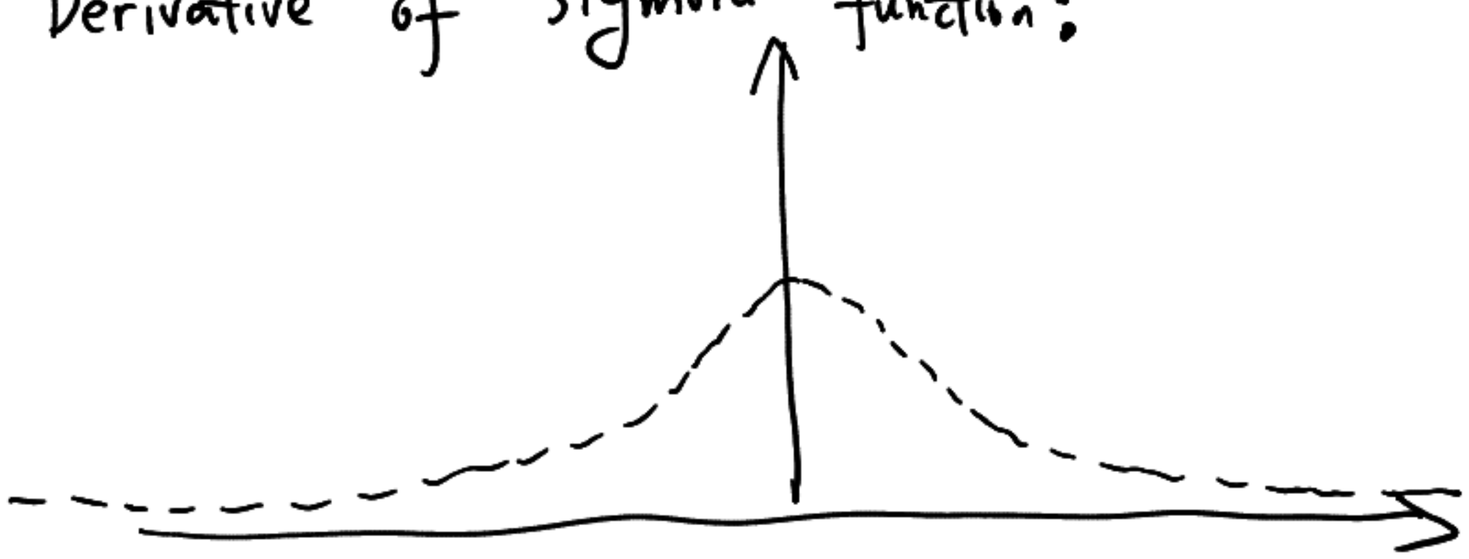


$$\sigma(z) = \frac{1}{1+\exp(-z)} \Rightarrow \sigma'(z) = \frac{-\exp(-z)\cdot(-1)}{(1+\exp(-z))^2}$$

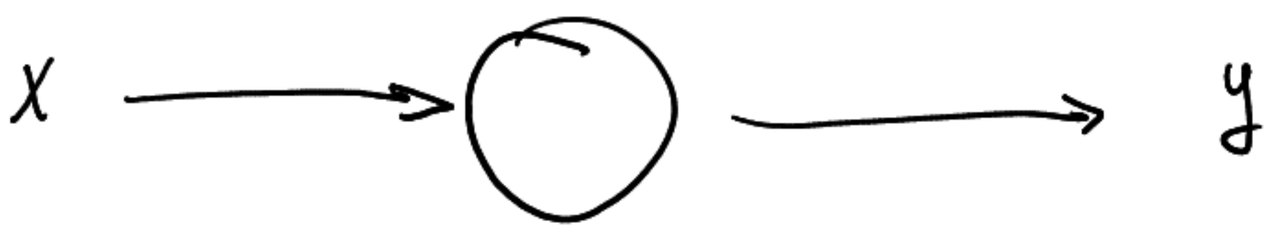
$$= \frac{\exp(-z)}{(1+\exp(-z))^2}$$

Ex. Verify the gradient of sigmoid function and the figure

Derivative of Sigmoid function :



Cross-entropy loss.



IN Sigmoid neuron OUT.

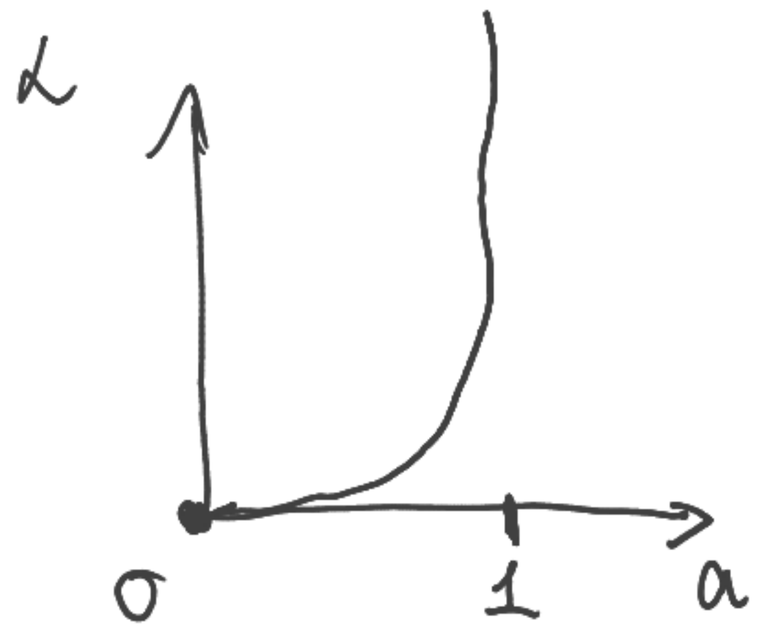
$$C(w,b) = -\left(y \cdot \ln(a) + (1-y)\cdot \ln(1-a)\right)$$

Facts about Cross-entropy loss

(i) $C(w,b) \geq 0$.

(ii) If a is close to y, then the loss is small.

— If $y=0$, then $\mathcal{L}(w,b) = -\ln(1-a)$.



Recall that a is the output of a sigmoid neuron, hence a always lies within 0 and 1.

— Ex. If $y=1$, verify the above is true

Learning slowdown at initialization.

Let's calculate the gradients of $\mathcal{L}(w,b)$ again.

$$\frac{\partial \mathcal{L}}{\partial w} = -\left(\frac{y}{a}\cdot\frac{\partial a}{\partial w} - \frac{1-y}{1-a}\cdot\frac{\partial a}{\partial w}\right)$$

$$= -\left(\frac{y}{a} - \frac{1-y}{1-a}\right)\cdot \sigma'(z)\cdot x$$

$$= -\frac{y-a\cdot y-(a-a\cdot y)}{a(1-a)}\cdot \sigma'(z)\cdot x$$

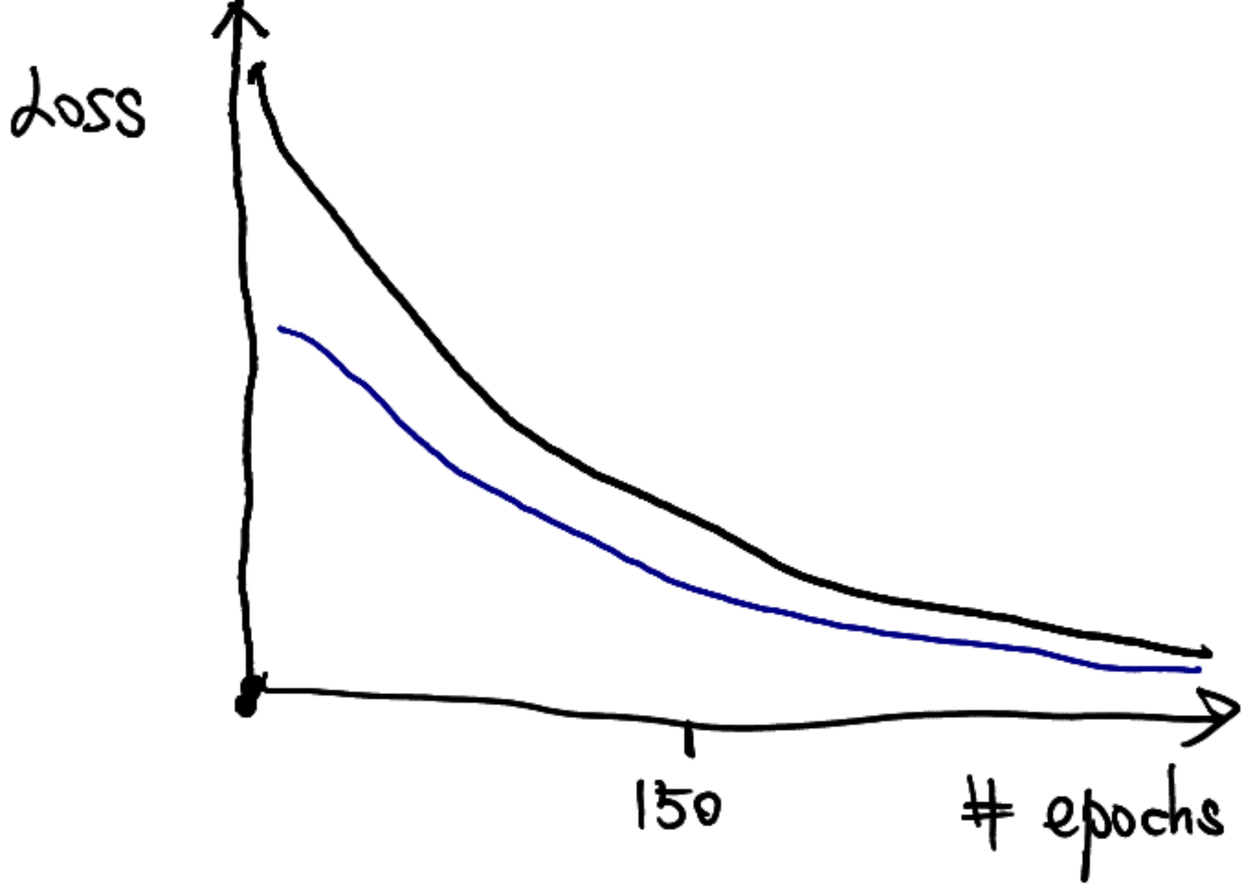
$$= (a-y)\cdot x\cdot\frac{\sigma'(z)}{a(1-a)}$$

$$= (a-y)\cdot x$$

Observation: The "$\sigma'(z)$" term went away compared to the quadratic loss!

Ex. $\dfrac{\sigma'(z)}{a(1-a)} = 1$

Recall that $a=\sigma(z)$



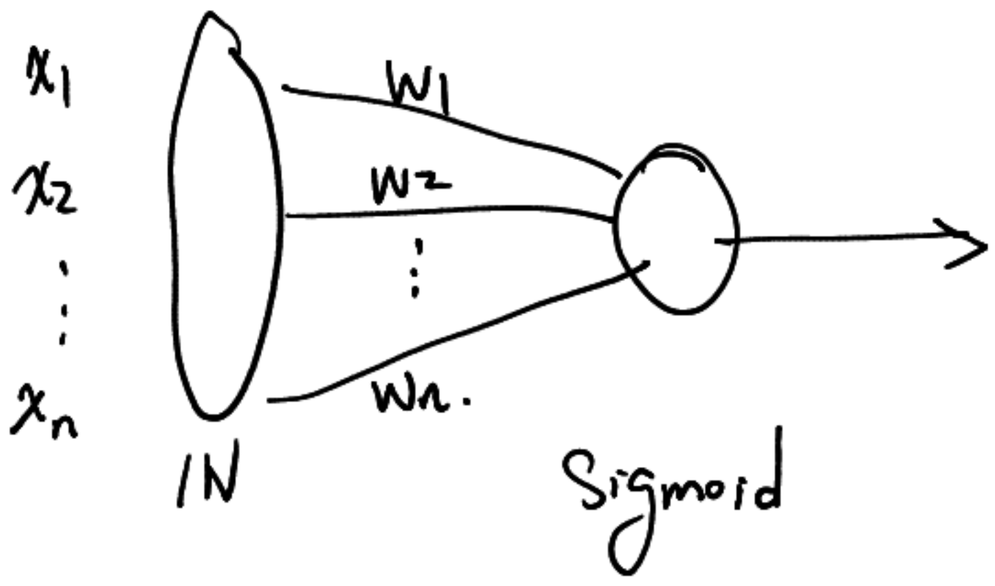
init : $w^{(0)} = 0.6$, $b^{(0)} = 0.9$

init : $w^{(0)} = 2$, $b^{(0)} = 2$.

Ex. Verify this phenomenon using Pytorch

So, the simulation indeed verified that using the cross-entropy loss, the learning process is faster when the error is large in the beginning!

Multi-dimensional Input.



$$a = \quad \sigma(\langle x, w \rangle + b)$$

We can extend the analysis above and show that

$$\frac{\partial \mathcal{L}}{\partial w_j} = \frac{1}{m} \sum_{x \in \text{Training Set}} x_j \left(\sigma(z) - y \right)$$

Going back to linear activation.

$$\sigma(z) = z$$

For quadratic loss, we have $\mathcal{L}(w,b) = \frac{(a-y)^2}{2}$

$$\Rightarrow \frac{\partial \mathcal{L}}{\partial w} = (a-y) \cdot \frac{\partial a}{\partial w} = (a-y) \cdot x .$$

So the quadratic loss will not give rise to any problems with a learning slowdown in this setting.

Deriving the cross-entropy loss function.

Imagine ~~that~~ we have discovered the learning slowdown phenomenon and understood ~~it~~ it is caused by the $\sigma'(z)$ factor. So, if we would like our gradients to behave as

$$\frac{\partial \mathcal{L}}{\partial w_j} = (a-y) \cdot w_j$$

$$\frac{\partial \mathcal{L}}{\partial b} = (a-y) ,$$

we can use them to derive the cross-entropy loss!

Using chain rule, we have that

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial b} = \frac{\partial L}{\partial a} \cdot \sigma'(z)$$

(Recall that $\sigma'(z) = \frac{\exp(-z)}{(1+\exp(-z))^2} = a \cdot (1-a)$)

$$= \frac{\partial L}{\partial a} \cdot a \cdot (1-a)$$

$$\Rightarrow \quad \frac{\partial L}{\partial a} \cdot a \cdot (1-a) = (a-y)$$

$$\Rightarrow \quad \frac{\partial L}{\partial a} = \frac{a-y}{a(1-a)} = -\frac{y}{a} + \frac{1-y}{1-a}$$

$$\Rightarrow \quad L = -y \cdot \ln a - (1-y) \cdot \ln(1-a) + \text{Constants}.$$

Summary. The loss is determined by the behavior of the gradients!

Multi-class Classification.

The sigmoid activation outputs a prob. value for the output, how about multi-class classification?

Softmax:

$$\Pr(\text{output class } i) \qquad a_i = \frac{\exp(z_i)}{\sum_{j=1}^{k} \exp(z_j)}.$$

Indeed one can verify that $\sum_{i=1}^{k} \Pr(\text{output class } i) = 1$.

What about the learning slowdown problem? Recall that for multi-class classification, we use negative log-likelihood loss.

Ex. Show that the learning slowdown problem does not occur using the negative log-likelihood loss over softmax activation.

3.3 Overfitting and regularization

— Overfitting

Overfitting is a major problem in neural networks. This is especially true in modern networks, which often have very large numbers of weights and biases.

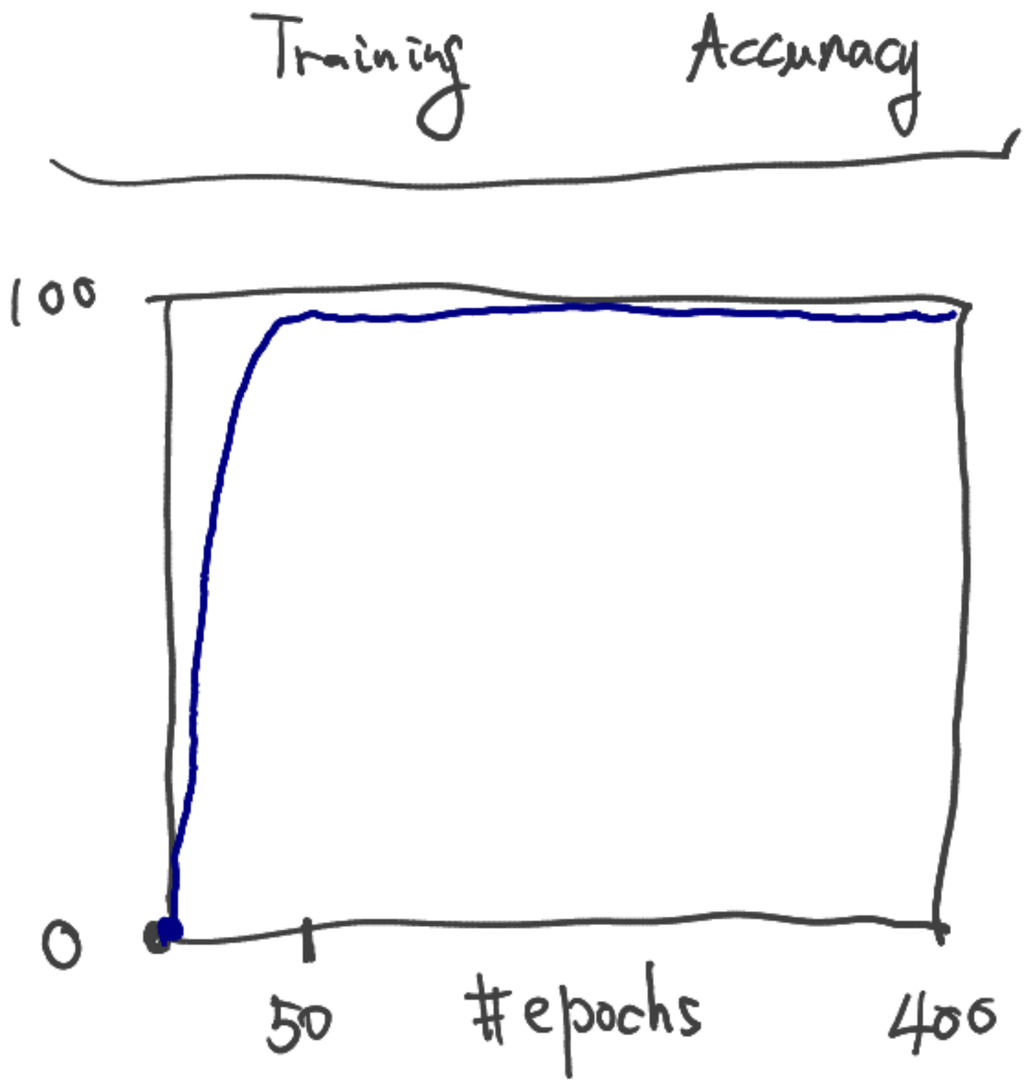
Ex. Suppose that we use a two-layer neural net with 30 hidden neurons. Instead of using the entire MNIST dataset, we'll use just 1,000 training images.

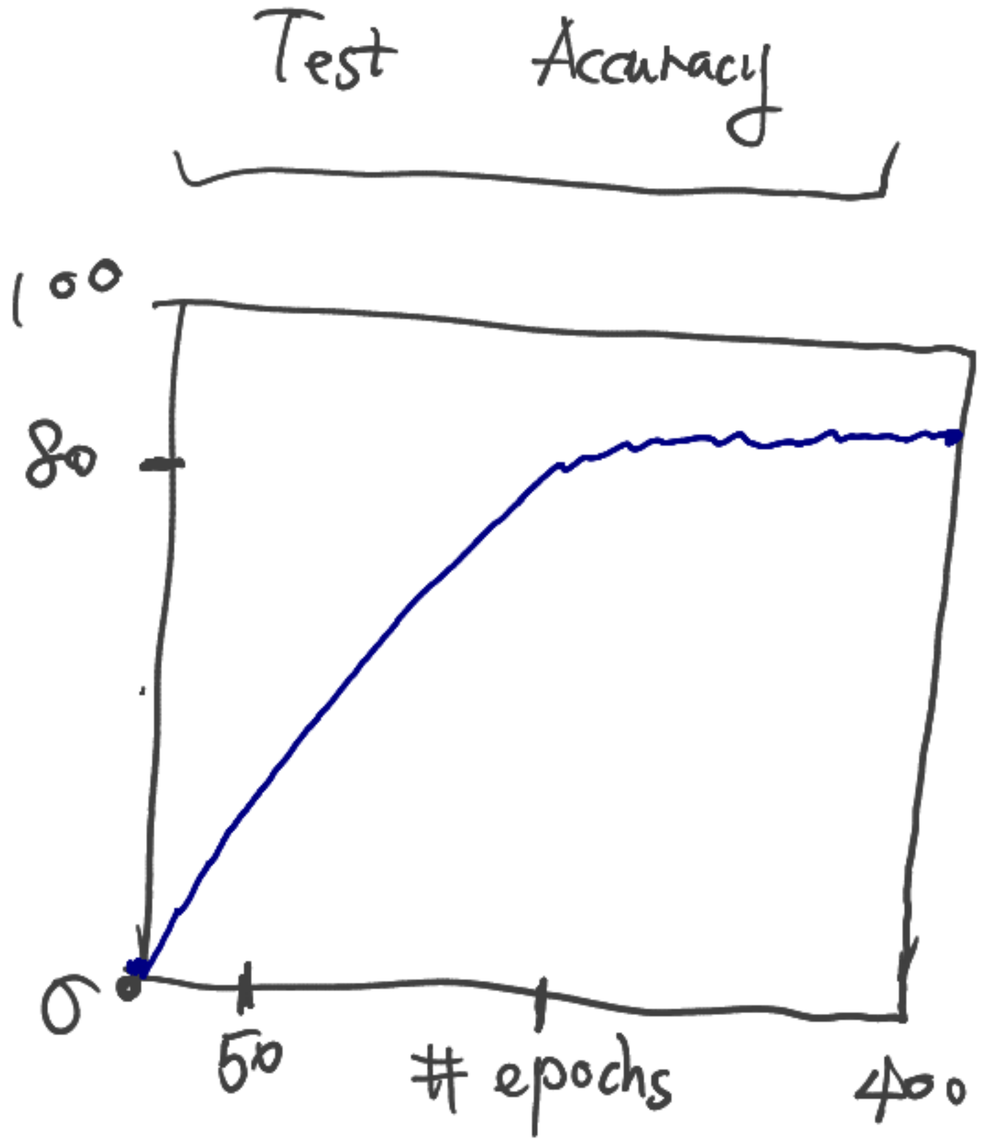
$$\#\text{ param} : 30 \times 28^2 \approx 24{,}000$$

$$\#\text{ samples} : 1{,}000$$

Let's see how it works!

Remark. # param $\gg$ # samples!





Remark. (i) Training accuracy goes to 100% quickly after 50 epochs.

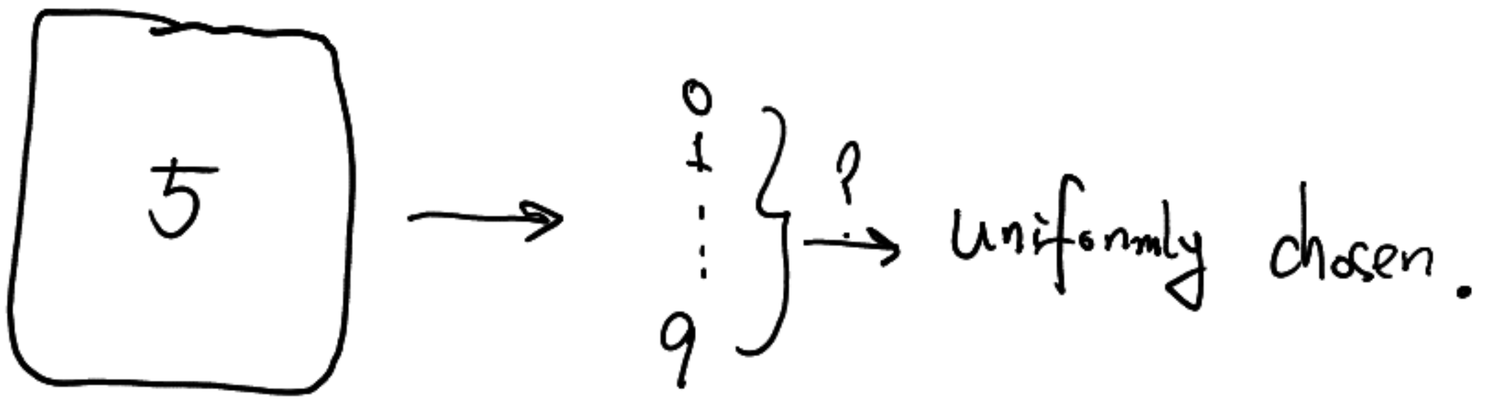
(ii) Test accuracy converges at ~ 82% accuracy.

(iii) Test accuracy increases even after training accuracy has saturated at 100% after 50 epochs ?!

Question. How come the test accuracy continues to improve even though the model has fit the training data perfectly ? This question has puzzled generalization theorists for several years. :C

The generalization mystery. [C. Zhang et al. ICLR'17]

Observation: Modern CNNs can fit real inges with random labels.



(iv). Even though the model has enough capacity to overfit or memorize the training ~~dataset~~, it still achieves reasonably good performance.

This question has inspired the area of studying generalization in over-parametrized models. There are two dominant trends in the literature:

(i) Data-dependent generalization bounds/measures:

— Rademacher complexity, Margin bounds

(ii) Implicit regularization: Inductive bias of the optimization algorithm, in particular (stochastic) gradient descent.

3.2 Regularization

- L2 regularization, a.k.a weight decay in neural network folks, and Ridge penalty in statistics.

For a traing loss $f = \sum_{(x,y)} \text{loss}(x,y) + \frac{\lambda}{2n} \sum_{i=1}^{d} \|W_i\|_F^2$

$\|W_i\|_F^2$ = sum of squares of the entries of W_i.

let's look at what happens after one step of gradient descent over f.

$$\nabla f = \sum_{(x,y)} \nabla \text{loss}(x,y) + \frac{\lambda}{n} \sum_{i=1}^{d} W_i .$$

$$\Rightarrow \quad W_i \leftarrow W_i - \eta \cdot \frac{\partial \left(\sum \text{loss}(x,y)\right)}{\partial W_i} - \frac{\lambda}{n} \cdot \eta \cdot W_i$$

$$= \left(1 - \eta \cdot \frac{\lambda}{n}\right) W_i - \eta \cdot \frac{\partial \left(\sum \text{loss}(x,y)\right)}{\partial W_i}$$

Hence the L2 regularization term "decays" W_i by an $\eta \cdot \frac{\lambda}{n}$ fraction!

Why is L2 regularization useful for preventing overfitting?

Occam's Razor: Smaller weights are, in some sense, lower complexity, and so provides a simpler and more powerful explanation for the data, and should be preferred.

Case study. Ridge regression and the bias-variance tradeoff

While this is not strictly a neural network per se, we might still get valuable insight!

Setup: We have n labeled data

$$(x_1, y_1), \quad (x_2, y_2) \quad \cdots, \quad (x_n, y_n).$$

$x_i \in \mathbb{R}^p$, $y_i \in \mathbb{R}$, and $y_i = x_i^T \beta + \varepsilon_i$,

for a fixed $\beta \in \mathbb{R}^p$ and $\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$.

Quadratic loss and the linear estimator:

$$L(w) = \sum_{i=1}^{n} (x_i^T w - y_i)^2$$

$$\nabla L(w) = 0 \Rightarrow w = (X^T X)^{-1} X^T Y$$

Ex. Verify the above equation

Prediction loss of w:

$$\mathcal{L}(w) = \mathop{\mathbb{E}}_{(x,y)} (x^T w - y)^2$$

$$= \mathop{\mathbb{E}}_{x} (x^T w - x^T \beta)$$

$$= (w-\beta)^T \underbrace{\mathbb{E}[xx^T]}_{\Sigma} \cdot (w-\beta).$$

$$= (w-\beta)^T \Sigma (w-\beta).$$

Recall that $w = (X^TX)^{-1}X^Ty = (X^TX)^{-1}X^T(X\beta + \varepsilon)$

$$= \beta + (X^TX)^{-1}X^T\varepsilon.$$

$$\Rightarrow \quad \mathcal{L}(w) = \varepsilon^T X (X^TX)^{-1} \Sigma (X^TX)^{-1} X^T \varepsilon.$$

Bias-Variance Tradeoff.

We can decompose $\mathcal{L}(w)$ into two parts.

$$\mathcal{L}(w) = \mathop{\mathbb{E}}_{x} \left(x^T w - y \right)^2$$

$$= \mathop{\mathbb{E}}_{x} \left(x^T \left(w - \mathop{\mathbb{E}}_{w}[w] + \mathop{\mathbb{E}}_{w}[w] \right) - y \right)^2$$

$$= \underbrace{\mathop{\mathbb{E}}_{x} \left(x^T \left(w - \mathop{\mathbb{E}}_{w}[w] \right) \right)^2}_{\text{Variance of } w} + \underbrace{\mathop{\mathbb{E}}_{x} \left(\mathop{\mathbb{E}}_{w}[w] - y \right)^2}_{\text{Bias of } w}$$

Ex. Verify that the cross-term is 0

Claim. Variance reduces after adding the $\ell 2$ regularization.

Proof. $\text{Variane} = \underset{x}{\mathbb{E}}\left[x^T (w - \underset{w}{\mathbb{E}}[w]) \right]^2$

Part 1. $w = \beta + (X^T X)^{-1} X^T \varepsilon$.

$\Rightarrow \mathbb{E}[w] = \beta$

$\Rightarrow \quad w - \underset{w}{\mathbb{E}}[w] = (X^T X)^{-1} X^T \varepsilon$.

Hence, $\text{Variance}(w) = \underset{x}{\mathbb{E}} \underset{\varepsilon}{\mathbb{E}} \left[x^T (X^T X)^{-1} X^T \varepsilon \right]$

$= \sigma^2 \cdot \text{Tr}\left[\Sigma (X^T X)^{-1} \right]$ ← Ex. Verfy this equation

Part 2. w_{ridge} ?

$$\ell(w) = \frac{1}{n} \sum_{i=1}^{n} (x_i^T w - \beta)^2 + \lambda \cdot \|w\|^2$$

$\nabla \ell(w) = 0 \Rightarrow$

$$w_{ridge} = (X^T X + \lambda \cdot Id)^{-1} X^T Y$$

$$= (X^T X + \lambda \cdot Id)^{-1} X^T (X\beta + \varepsilon)$$

$$\Rightarrow w_{ridge} - \underset{w_{ridge}}{\mathbb{E}}[w_{ridge}] = (X^T X + \lambda \cdot Id)^{-1} X^T \varepsilon$$

$$\Rightarrow \text{Variance}(w_{ridge}) = \sigma^2 \cdot \text{Tr}\left[\Sigma \cdot (X^T X + \lambda \cdot Id)^{-2} \cdot X^T X\right]$$

$$< \text{Variance}(w), \quad \text{because of the } \lambda \cdot Id \text{ term above!}$$

To summarize, the ridge regularization reduces the variance of the estimator by "shrinking" the weights.

- L1 regularization, a.k.a. lasso in statistics.

In this approach, we modify the unregularized cost function by adding the sum of the absolute values of the weights:

$$J = \frac{1}{n} \sum \text{Loss}(x, y) + \lambda \cdot \sum_i \| W_i \|_1,$$

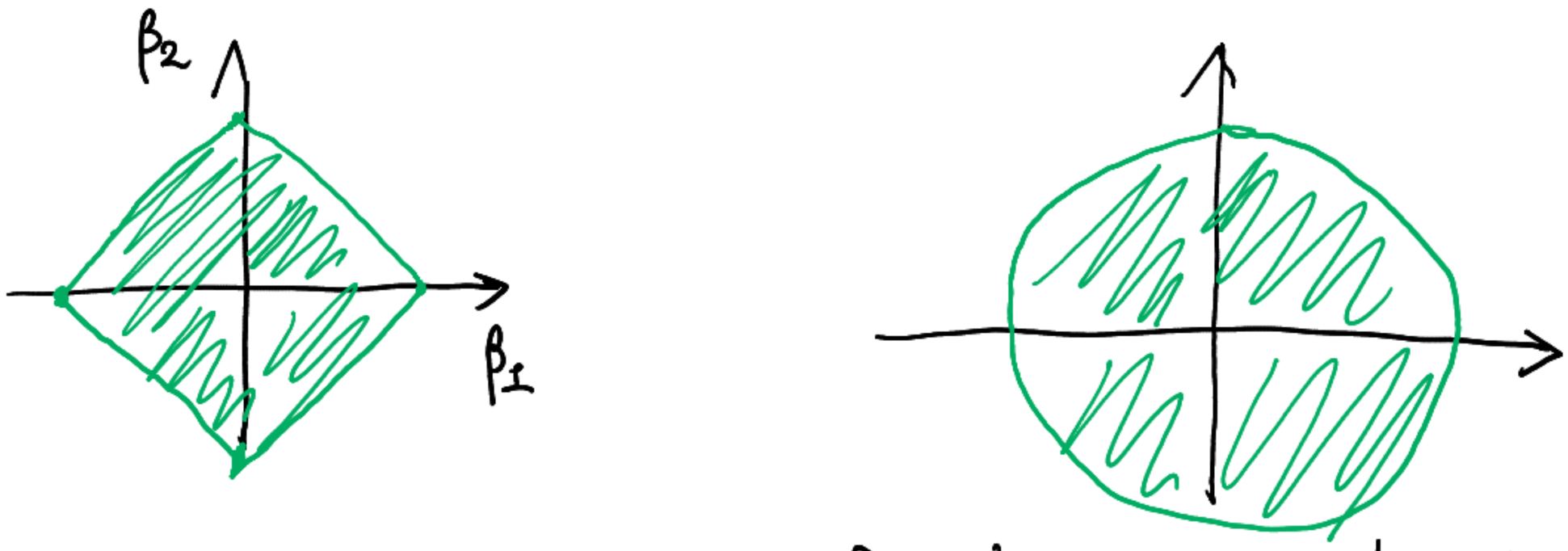
$\| W_i \|_1$ = sum of the absolute values of the weights.

Remark. L1 regularization is useful for feature selection.

There is a very interesting paper from Behnam Neyshabur that shows MLP + L1 regularization achieves competitive performance on CIFAR!

Geometric comparison between $L2$ and $L1$ regularization:

$L1: |\beta_1| + |\beta_2| \le t$ $\qquad$ $L2: \beta_1^2 + \beta_2^2 \le t$



Let's look at the derivatives of the $L1$ regularization.

$$\frac{\partial f}{\partial w_i} = \frac{1}{n}\frac{\partial Loss}{\partial w_i} + \lambda \cdot sgn(w_i),$$

where $sgn(x) = \begin{cases} +1, & \text{if } x > 0 \\ -1, & \text{if } x < 0 \end{cases}$

$$\Rightarrow w_i \leftarrow w_i - \eta \cdot \nabla \frac{\partial f}{\partial w_i}$$

$$= w_i \cdot \left(1 - \frac{\eta \cdot \lambda}{n}\right) - \eta \cdot \frac{\partial Loss}{\partial w_i}$$

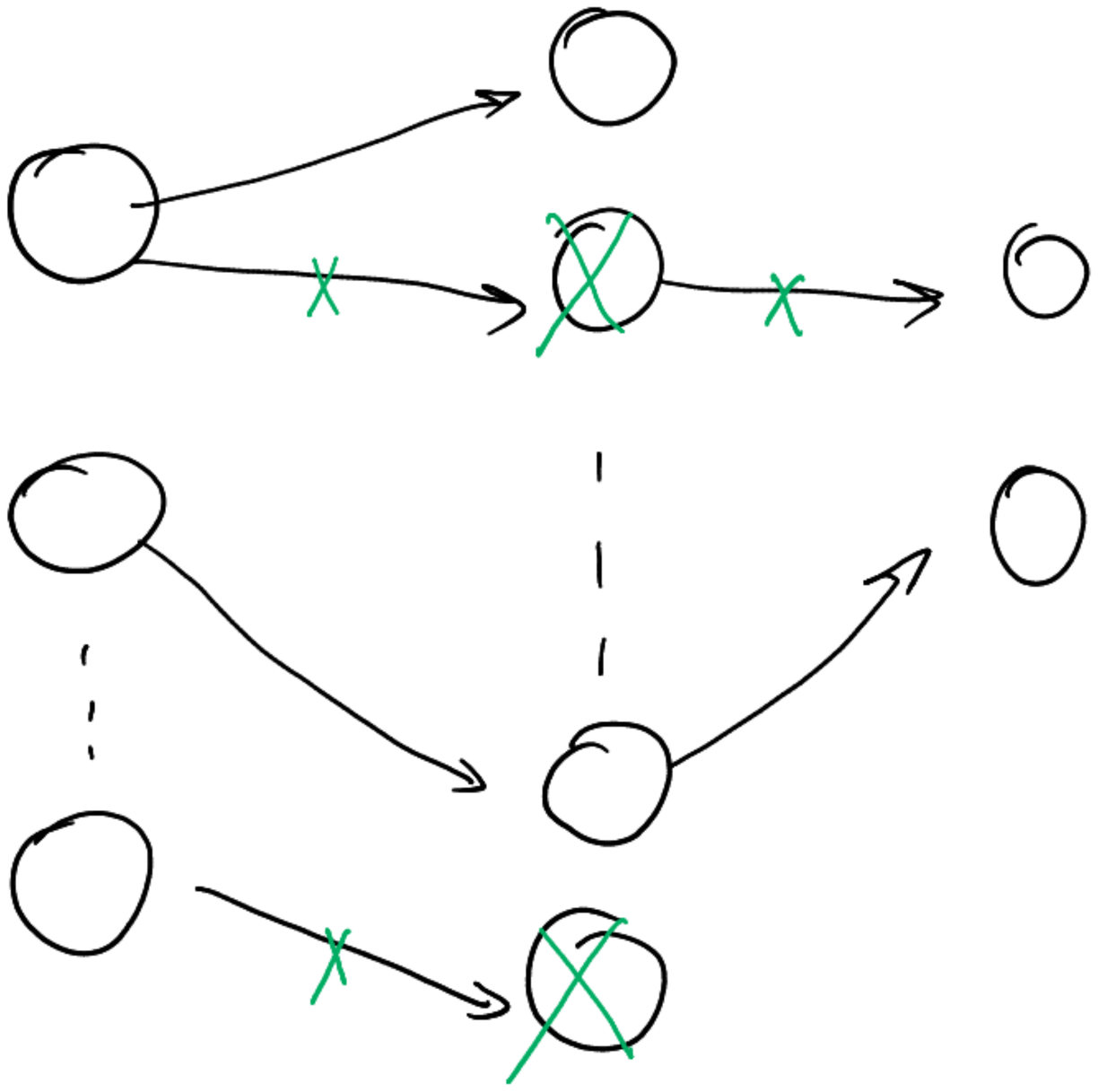
Compared to $L2$ regularization!

(Recall: $w_i \xleftarrow{L2} w_i \left(1 - \eta \cdot \frac{\lambda}{n} \cdot w_i\right) - \eta \cdot \frac{\partial Loss}{\partial w_i}$)

① When w_i is large, $L2$ shrinks more than $L1$.

② When w_i is small, $L1$ shrinks more than $L2$.

— Dropout



During every mini-batch update, we are going to randomly "delete" a fraction of neurons.

— Perform forward and backward pass after "deleting" a fraction of neurons.

— The neurons ~~that~~ got deleted are still present and may be used in the next mini-batch update.

Intuition: Dropout reduces the variance of the predictions of the individual neurons.

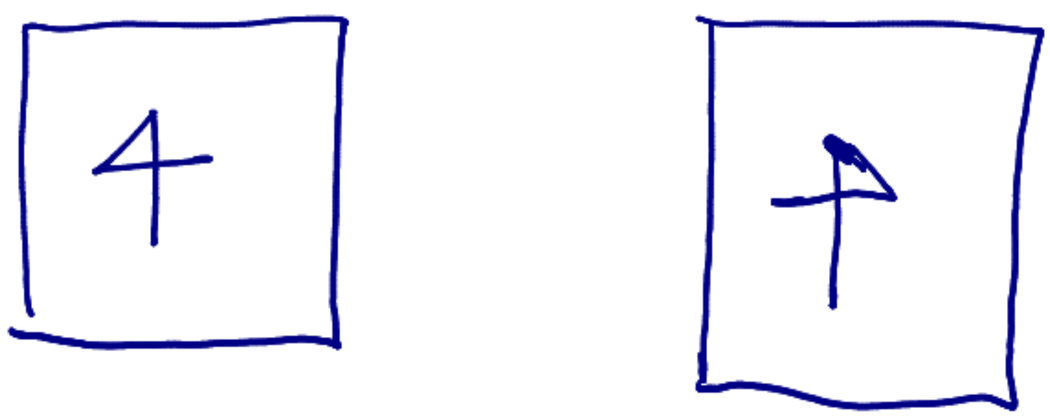
— Data Augmentation

Data augmentation is a technique for expanding the training dataset by transforming the available data.

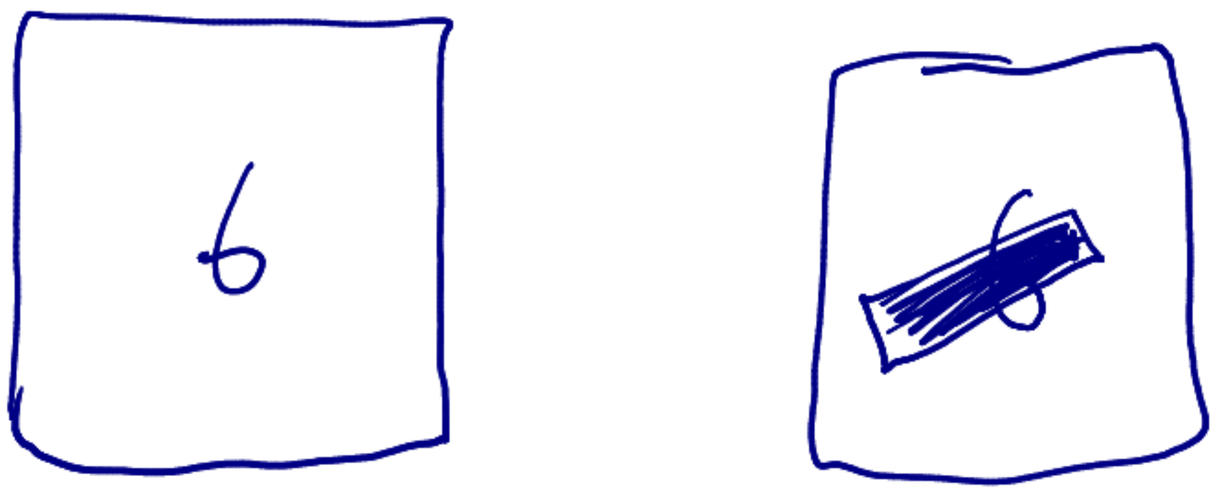
Ex.

Rotation.

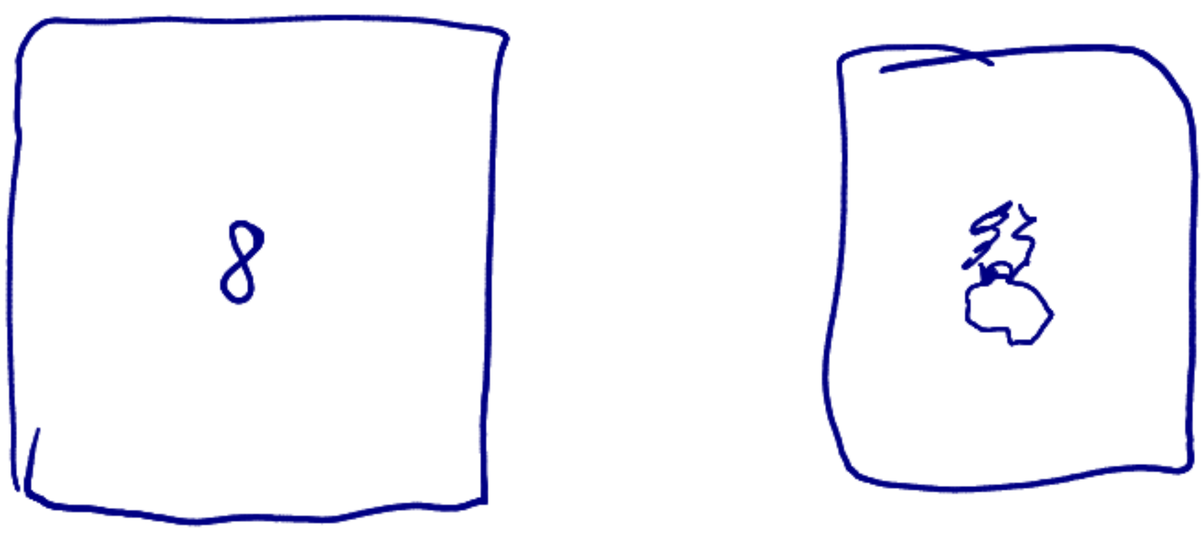
Horizontal flip.



Cutout.



Gaussian noise.



Conclusions.

(i) Overfitting is a significant challenge in modern neural networks.

(ii) Surprisingly, using a larger model does not necessarily lead to worse generalization.

- Generalization mystery.

(iii) L2 regularization : For linear regression, adding L2 reduces variance

(iv) L1 regularization : Encourages sparse solutions.

(v) Dropout : reduces variance among neurons.

(vi) Data Augmentation