SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Introduction to Neural Networks II

PROF LIM KWAN HUI

50.021 Artificial Intelligence

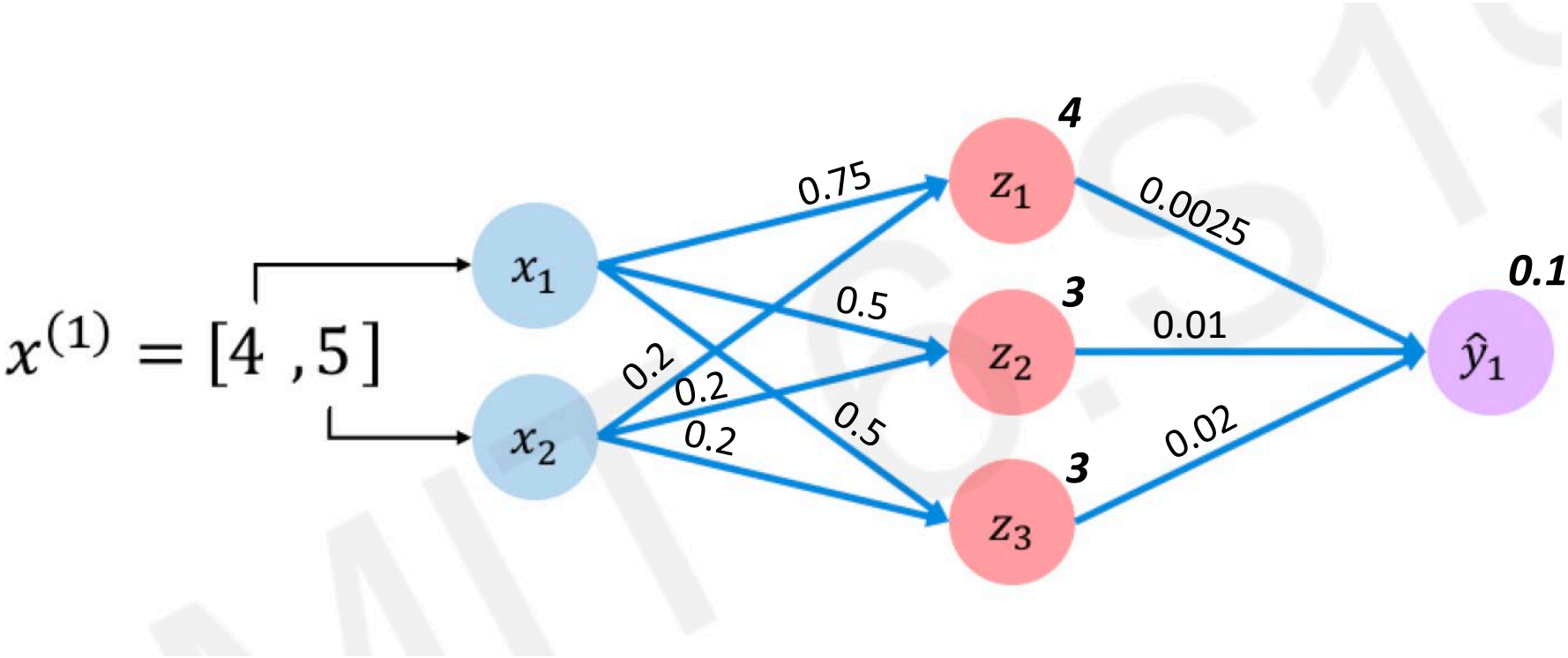
The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.

Outline & Objectives

- Understand how a perceptron works in the context of neural networks
- Understand how to train and use a neural network, including the general intuition behind activation functions, losses and optimizers
- Be aware of techniques to prevent overfitting in neural networks
- Be able to use a simple neural network to perform a certain task, e.g., compute the output for a prediction

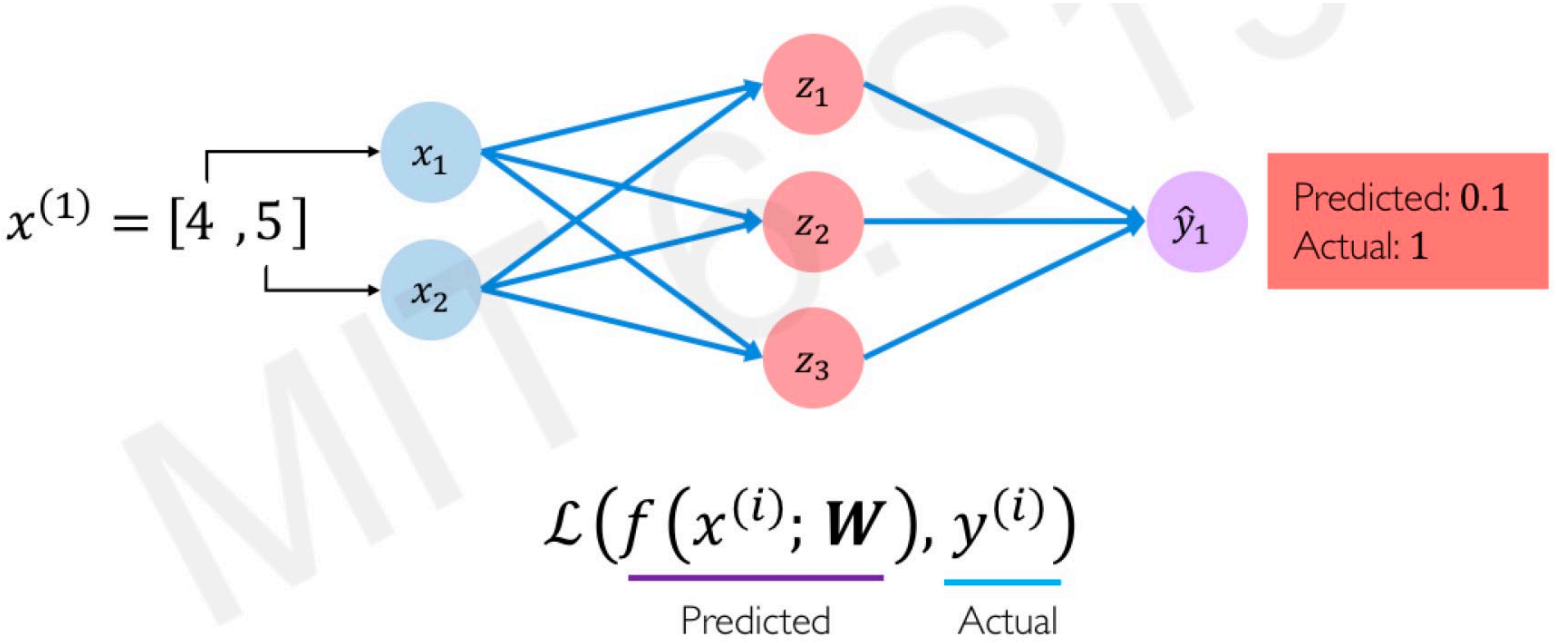
Recap: Will I pass this class?

- Compute the value at y_1 based on the weights given. Ignore the bias term and assume a linear activation g(z)=z.



Recap: Quantifying loss

- The **loss** of our network measures the cost from mispredictions



Training

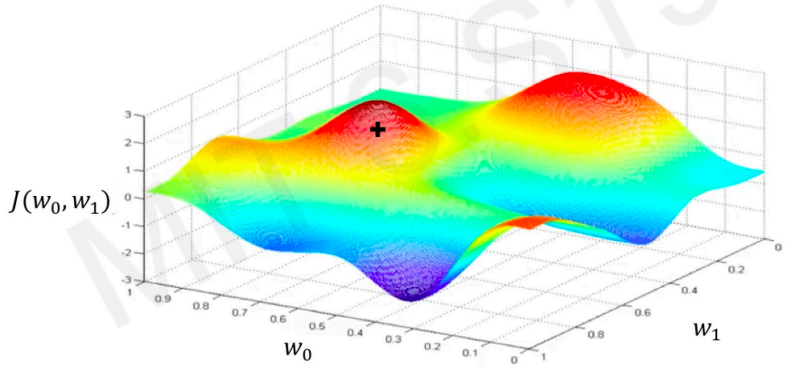
- Finding the weights that achieve the lowest loss
- Loss is a function of the network weights W!

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

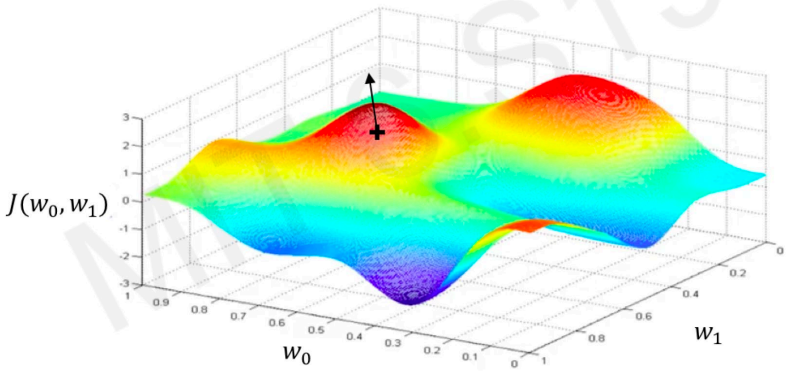
$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} J(\boldsymbol{W})$$

Loss optimization

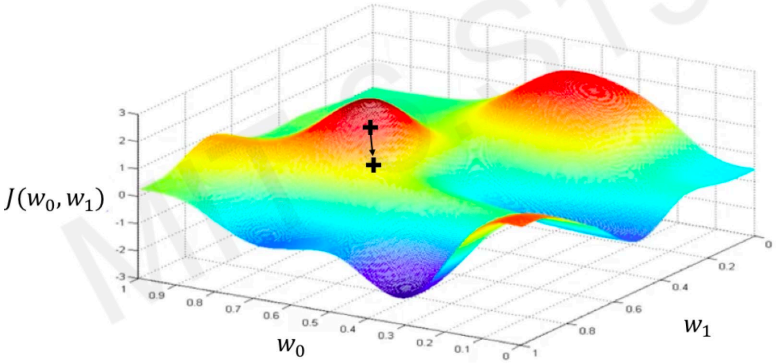
(1) Randomly pick an initial (w_0, w_1)



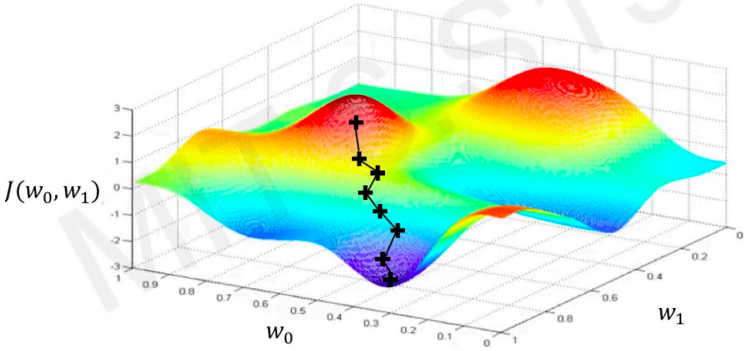
(2) Compute gradient, $\frac{\partial J(W)}{\partial W}$



(3) Take small step in opposite direction of gradient



(4) Repeat until convergence



Gradient descent

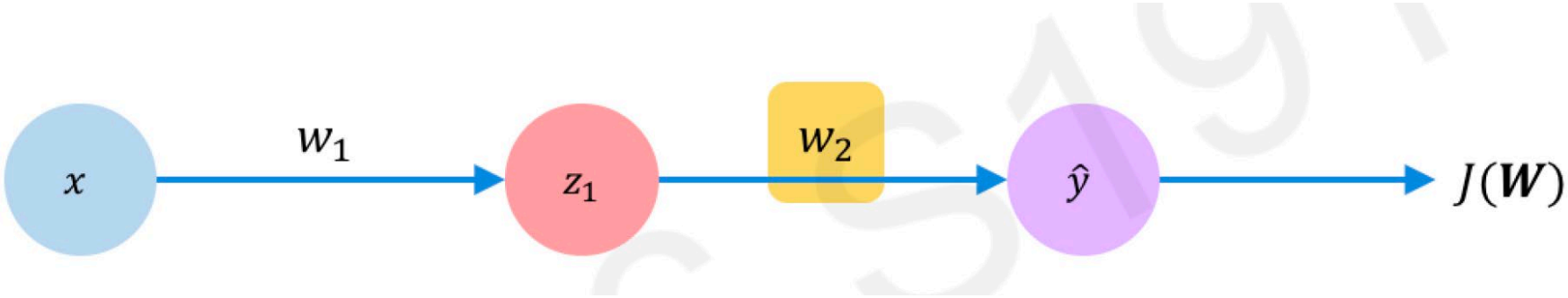
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
   1. Compute gradient $\frac{\partial J(W)}{\partial W}$
   2. Update weights $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
3. Return weights

```
optimizer =
optim.SGD(model.parameters(), lr =
0.01)

for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
```

Computing gradients: backpropagation

- How does a small change in weights (e.g. w_2) affect the final loss J(W)?

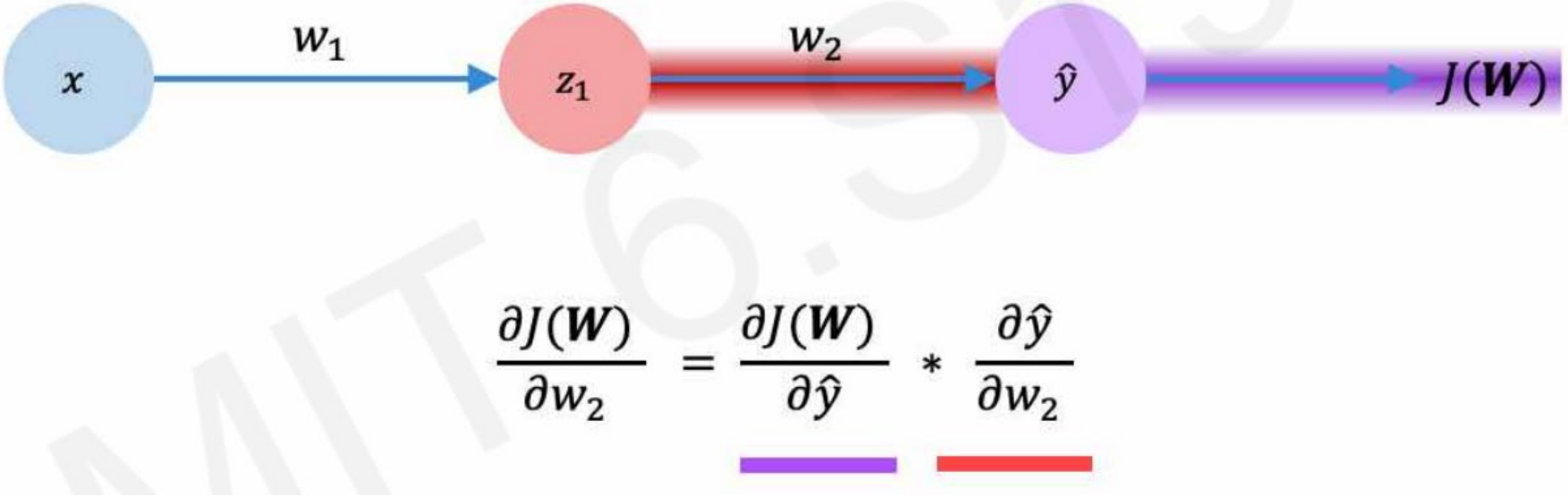


$$\frac{\partial J(\boldsymbol{W})}{\partial w_2} =$$

Let's use the chain rule!

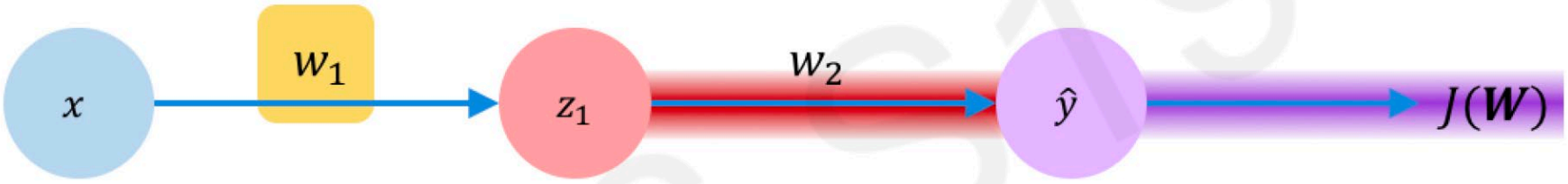
Computing gradients: backpropagation

- Repeat this for every weight in the network using gradients from later layers:



Exercise: Computing gradients - backpropagation

- How does a small change in weights (e.g. w_1) affect the final loss J(W)?

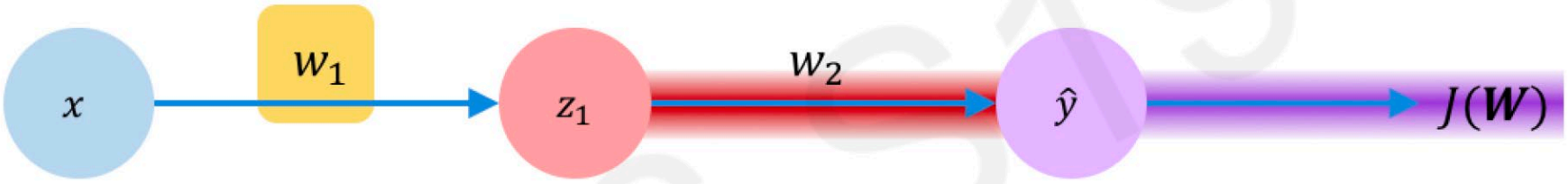


$$\frac{\partial J(W)}{\partial w_1} = ?$$

Apply chain rule!

Exercise: Computing gradients - backpropagation

- How does a small change in weights (e.g. w_1) affect the final loss J(W)?



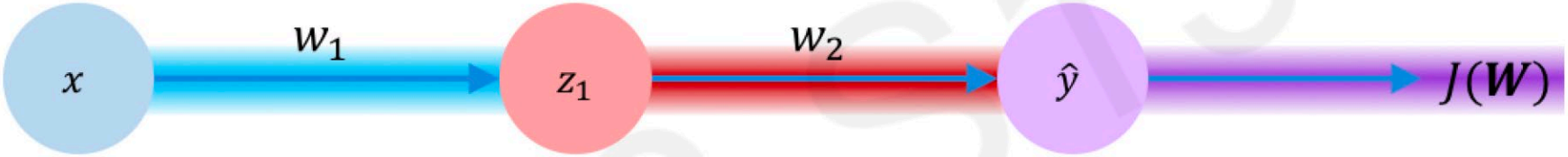
$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Apply chain rule!

Apply chain rule!

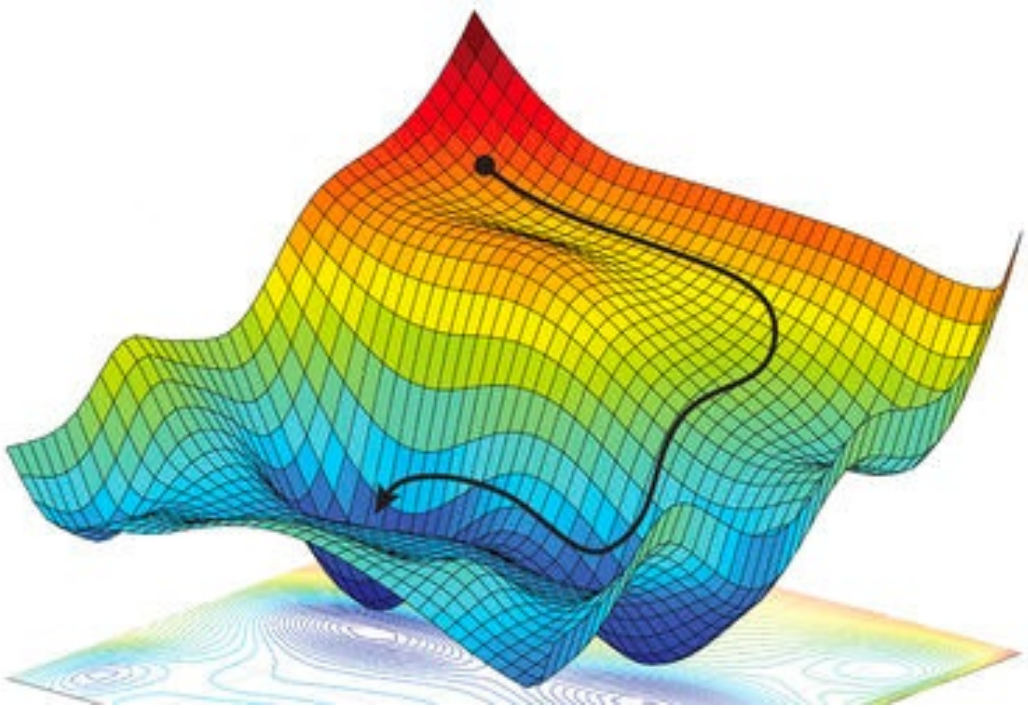
Exercise: Computing gradients - backpropagation

- Repeat this for every weight in the network using gradients from later layers:



$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

Optimizing the training



Gradient descent

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
   1. Compute gradient $\frac{\partial J(W)}{\partial W}$ **<— Hard to compute over the whole dataset**
   2. Update weights $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
3. Return weights

Gradient descent

- Easy computation.
  - Easy to implement.
  - Easy to understand.

algorithm: $\boldsymbol{\theta = \theta - \alpha \cdot \nabla J(\theta)}$

- Disadvantages:
  - May trap at local minima, needs a convex function
  - Weights are changed after **calculating gradient on the whole dataset**. So, if the dataset is too large than this may take **years to converge** to the minima.
  - Requires **large memory** to calculate gradient on the whole dataset.

Mini-batches

- More accurate estimation of gradient
- Smoother convergence
- Allows for larger learning rates

- Faster training
- Can allow parallelization of computation

Stochastic gradient descent

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
   1. Pick a single data point!
   2. Compute gradient $\frac{\partial J_i(W)}{\partial W}$ **<— Easy to compute but very noisy (stochastic!)**
   3. Update weights $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
3. Return weights

Mini-batch Stochastic gradient descent

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
    1. Pick a batch B of datapoints
    2. Compute gradient $\frac{\partial J(W)}{\partial W} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(W)}{\partial W}$ **<— Fast to compute and a much better estimate of the true gradient**
    3. Update weights $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
3. Return weights

Stochastic gradient descent

- Variant of Gradient Descent
- Tries to update the model's parameters more frequently
- Advantages:
  - Frequent updates of model parameters hence, **converges in less time**.
  - Requires **less memory** as no need to store values of loss functions.
  - May get **new minima's**.
- Disadvantages:
  - **High variance in model parameters**.
  - May shoot up again even after achieving global minimum.
  - To get the same convergence as gradient descent needs to slowly reduce the value of learning rate.

Batches help!

Optimizing loss functions

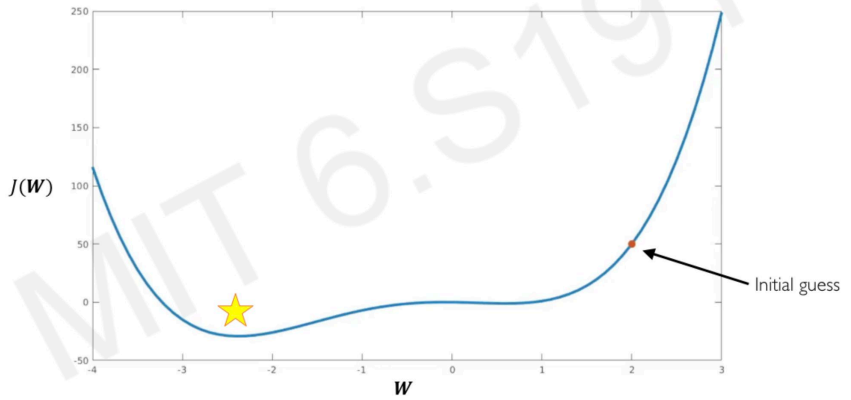
- Difficult!
- Small learning rate converges quickly (local optima)
- Large learning rate: unstable and diverges

Remember:

Optimization through gradient descent

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$$

How can we set the learning rate?



You want to find a stable learning rate that converges smoothly and avoids local optima

Optimizing loss functions

1. Try different learning rates…
2. Smarter: design an **adaptive learning** rate that adapts to the optimization landscape.
    1. Learning rate no longer fixed
    2. Can be made smaller or larger depending on:
        - How large gradient is
        - How fast learning is happening
        - Size of particular weights
        - …

Optimizers

: $\theta = \theta - \alpha \cdot \nabla J(\theta)$

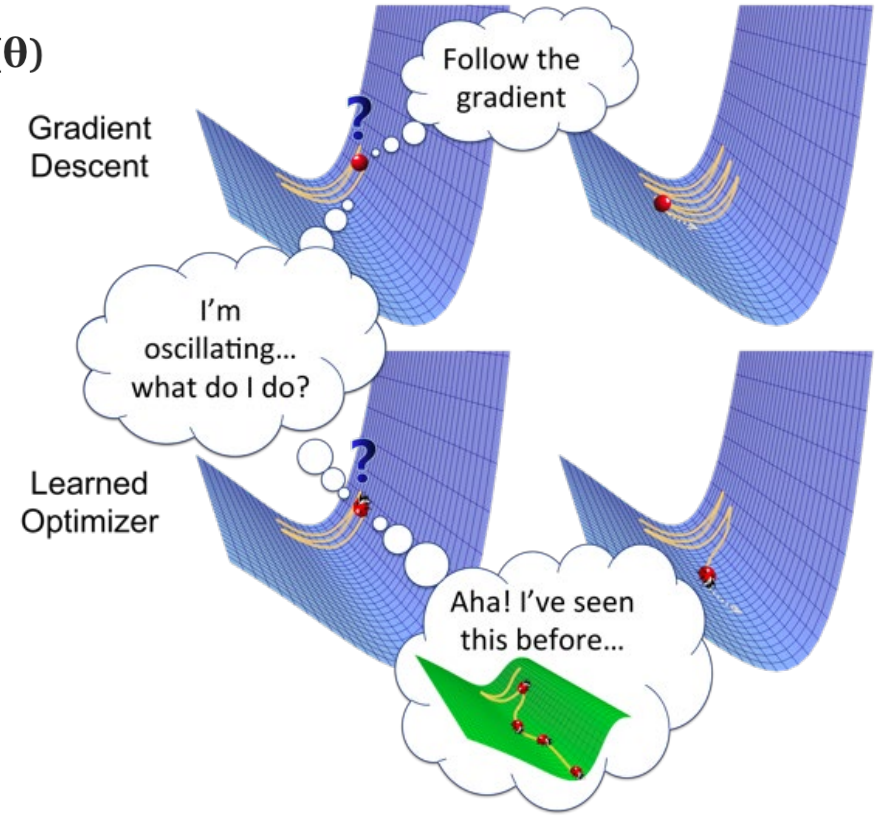


Image source: https://medium.com/@sdoshi579/optimizers-for-training-neural-network-59450d71caf6

Gradient descent algorithms

- Stochastic Gradient Descent (SGD)

```
torch.optim.SGD(input)
```

- Adam

```
torch.optim.Adam(input)
```

- Adagrad

```
torch.optim.Adagrad(input)
```

- Adadelta

```
torch.optim.Adadelta(input)
```

- RMSProp

```
torch.optim.RMSProp(input)
```

More on: https://pytorch.org/docs/stable/optim.html

Adagrad

- This optimizer **changes the learning rate**
- It changes the learning rate ‘η’ **for each parameter and at every time step ‘t’**.
- Advantages:
  - Learning rate changes for each training parameter.
  - Don’t need to manually tune the learning rate.
  - Able to train on sparse data.
- Disadvantages:
  - Computationally expensive
  - The learning rate is always decreasing results in slow training.

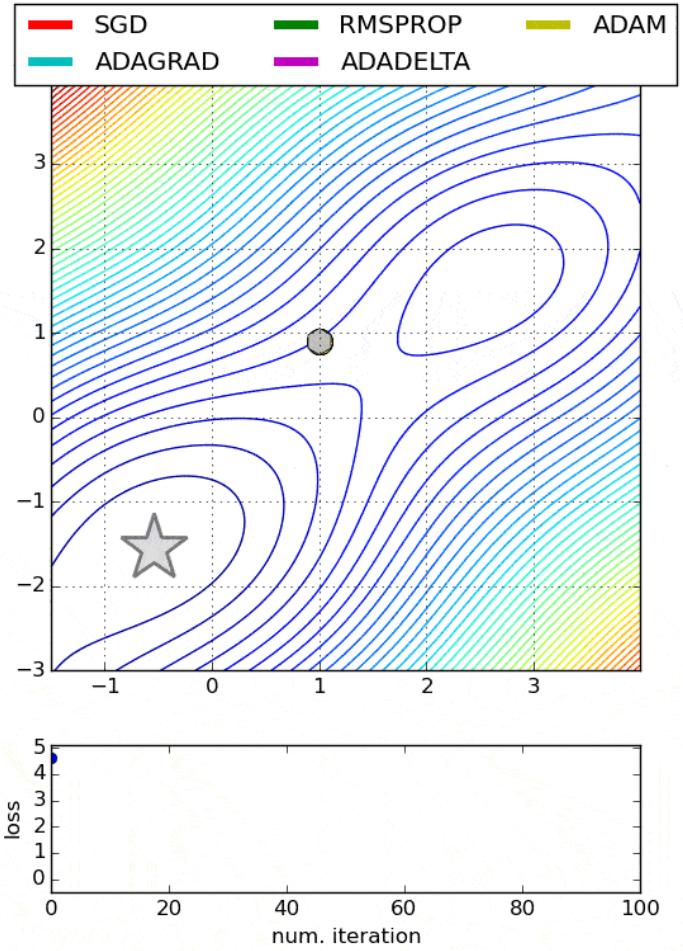
Adadelta

- Extension of AdaGrad
- Remove the decaying learning Rate problem
- Instead of accumulating all previously squared gradients, **Adadelta limits the window of accumulated past gradients to some fixed size w**. In this exponentially moving average is used rather than the sum of all the gradients.
- Advantages:
  - Now the learning rate does not decay and the training does not stop.
- Disadvantages:
  - Computationally expensive.

Adam

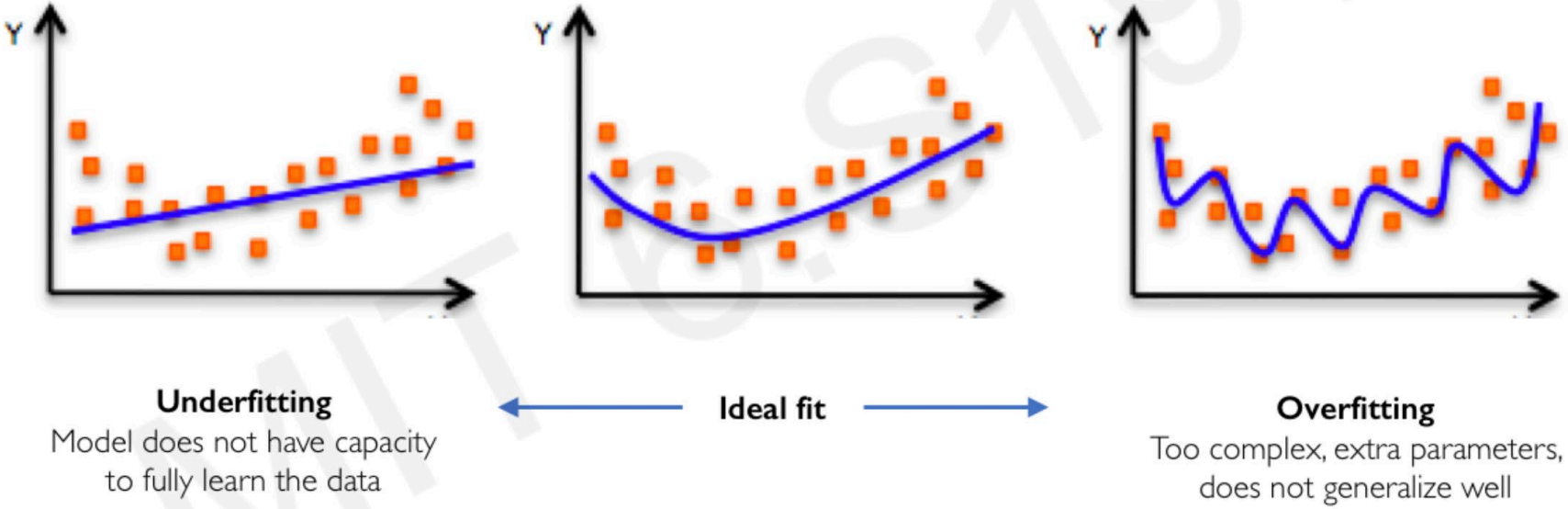
- Adaptive Moment Estimation (ADAM)
- The intuition behind the Adam is that **we do not want to roll so fast** just **because we can jump over the minimum**, we want to decrease the velocity a little bit for a careful search. In addition to storing an exponentially decaying average of past squared gradients like AdaDelta, Adam also keeps an exponentially decaying average of past gradients M(t).
- Advantages:
  - The method is very fast and converges rapidly.
  - Rectifies decaying gradients, high variance errors.
- Disadvantages:
  - Computationally costly.

Comparison of different optimizers



Source:
https://miro.medium.com/max/1200/1*_osB82GKHBOT8k1idLqiqA.gif

Overfitting neural networks (variance)



Overfitting neural networks

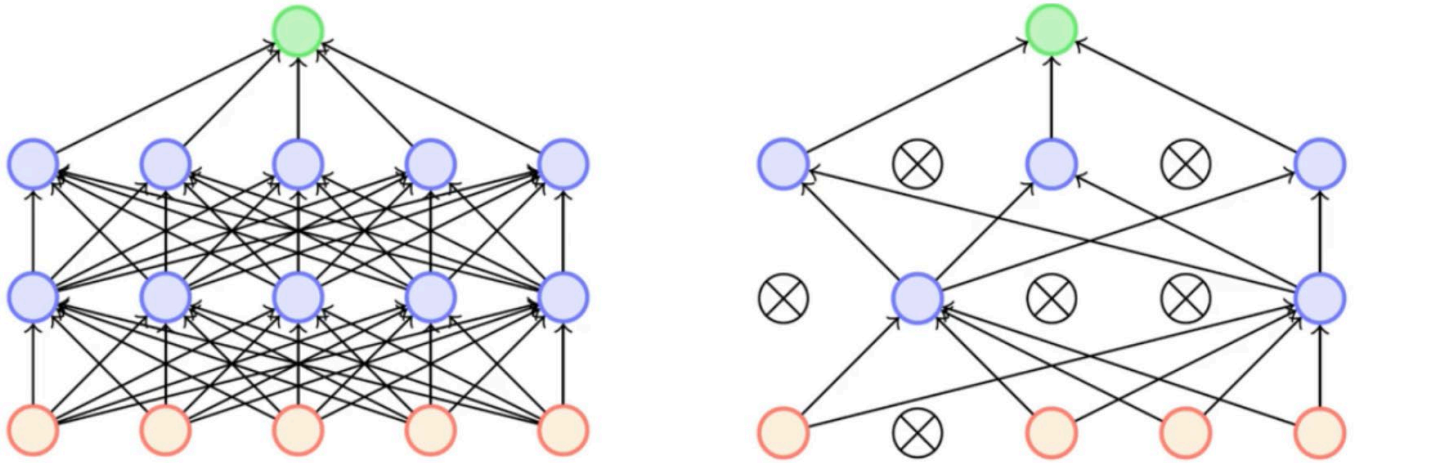
- Two often used remedies:
  - Reduce number of features
    - Manually select which features to keep
    - Different model
  - Regularization
    - Keep all features, but reduce the magnitude/size
    - Works well when we have a lot of features, each of which contributes a bit to predicting y

Regularization

- Technique to discourage complex models (high variance error)
- Why?
  - Improve generalization of our model on unseen data
- How?
  - Dropout
  - Early stopping
  - L1 & L2
  - Data augmentation?
    - Adds prior knowledge to a model;

Dropout

- During training, randomly set some activations to zero
  - Typically drop 50% of activations in a layer
  - Forces the network not to rely on any 1 node



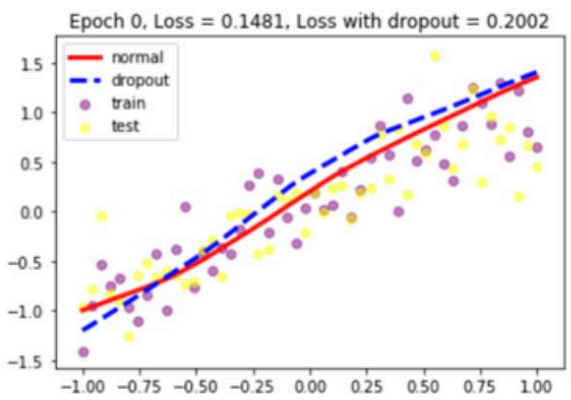
Original network

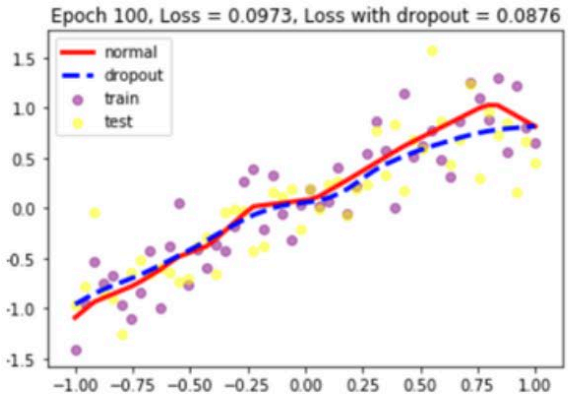
Network with some nodes dropped out

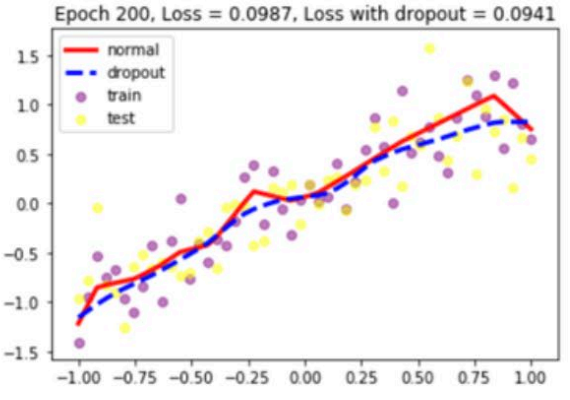
Image Source: https://towardsdatascience.com/batch-normalization-and-dropout-in-neural-networks-explained-with-pytorch-47d7a8459bcd

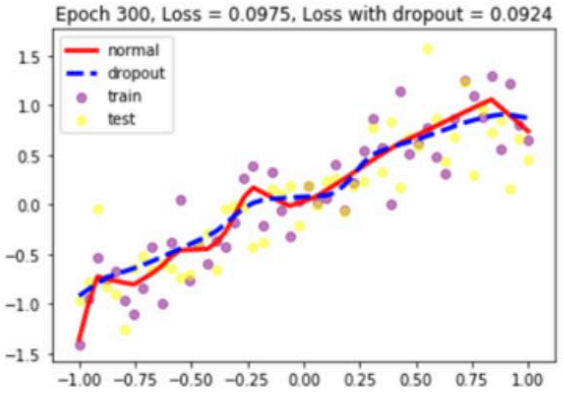
```
self.drop_layer = torch.nn.Dropout(p=0.5)
```

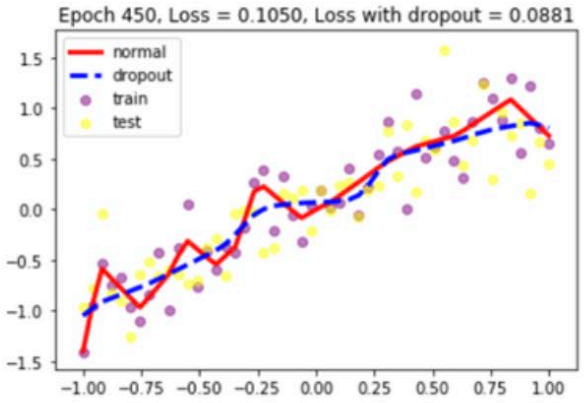
Dropout example

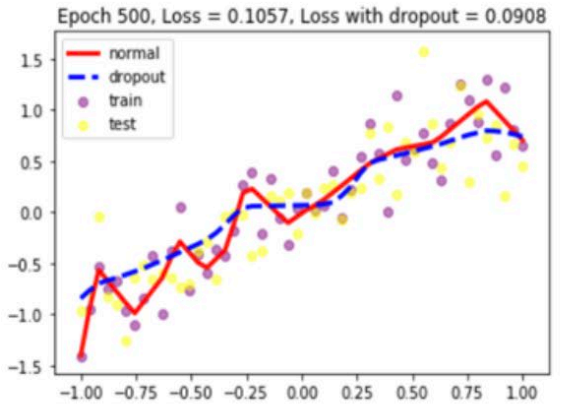






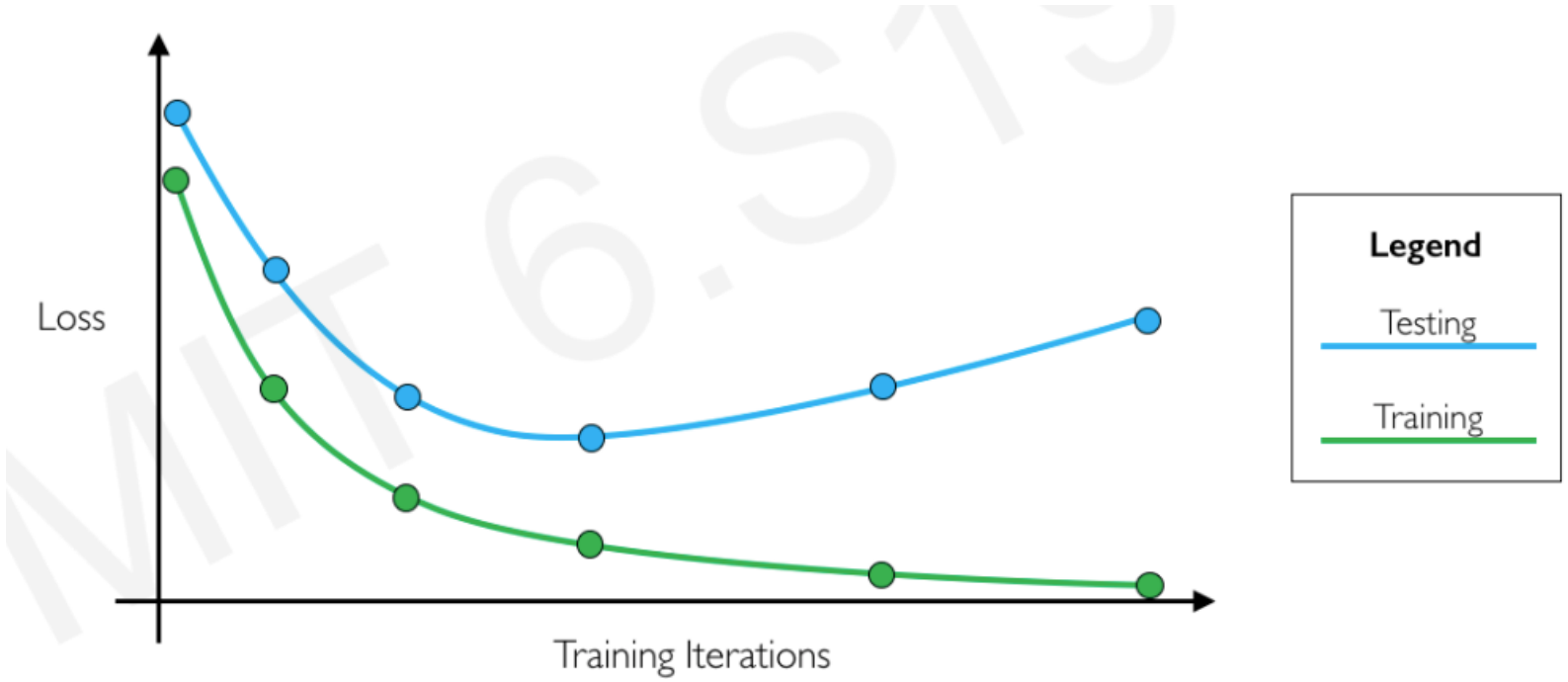






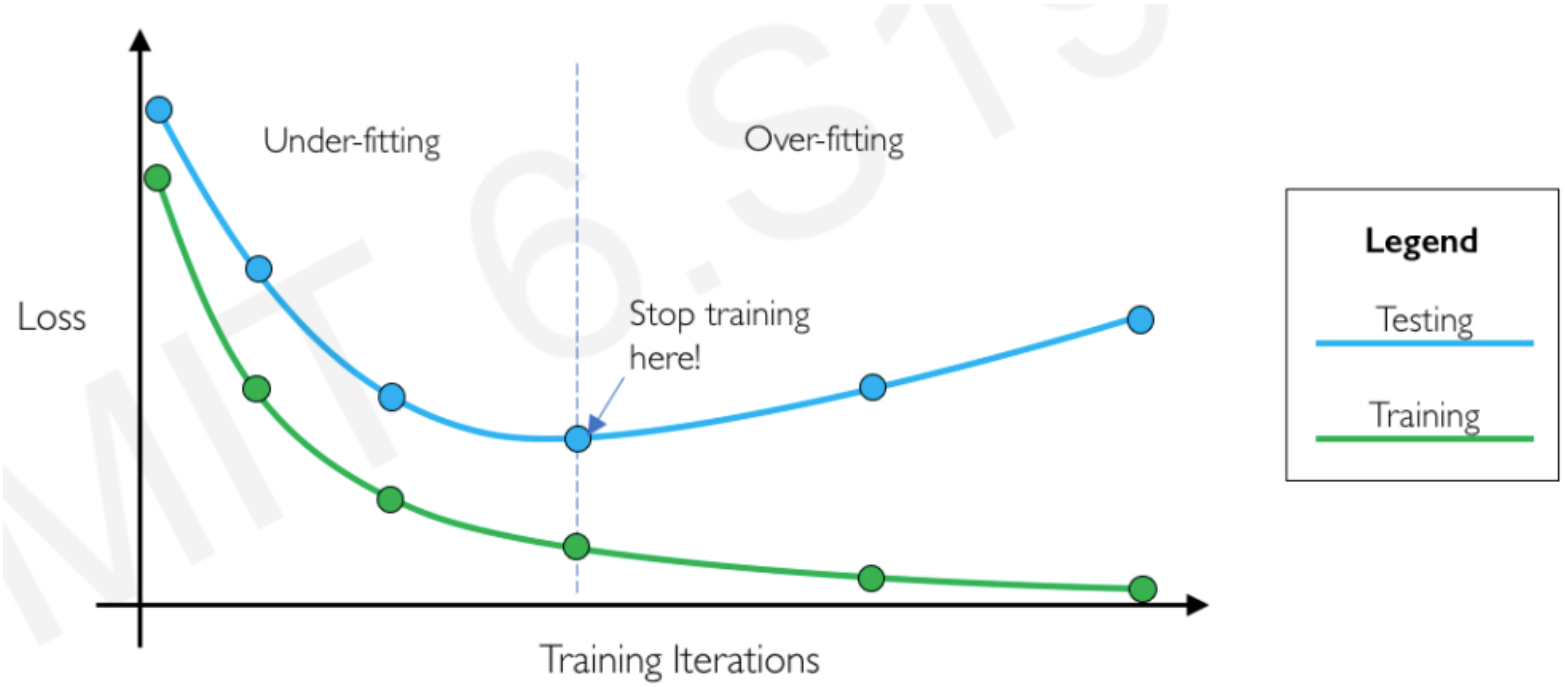
Early stopping

- Stop training before we have a chance to overfit



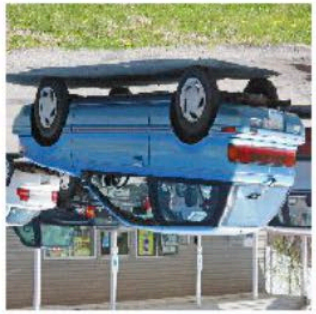
Early stopping

- Stop training before we have a chance to overfit



Data augmentation




```
final_train_data = []
final_target_train = []
for i in tqdm(range(train_x.shape[0])):
    final_train_data.append(train_x[i])
    final_train_data.append(rotate(train_x[i], angle=45, mode = 'wrap'))
    final_train_data.append(np.fliplr(train_x[i]))
    final_train_data.append(np.flipud(train_x[i]))
    final_train_data.append(random_noise(train_x[i],var=0.2**2))
    for j in range(5):
        final_target_train.append(train_y[i])
```

Types of regularisation

- L1 regularisation (Lasso or L1 norm)

$$L(x, y) \equiv \sum_{i=1}^{n} (y_i - f(x_i; w_i))^2 + \lambda \sum_{i=1}^{p} |w_i|$$

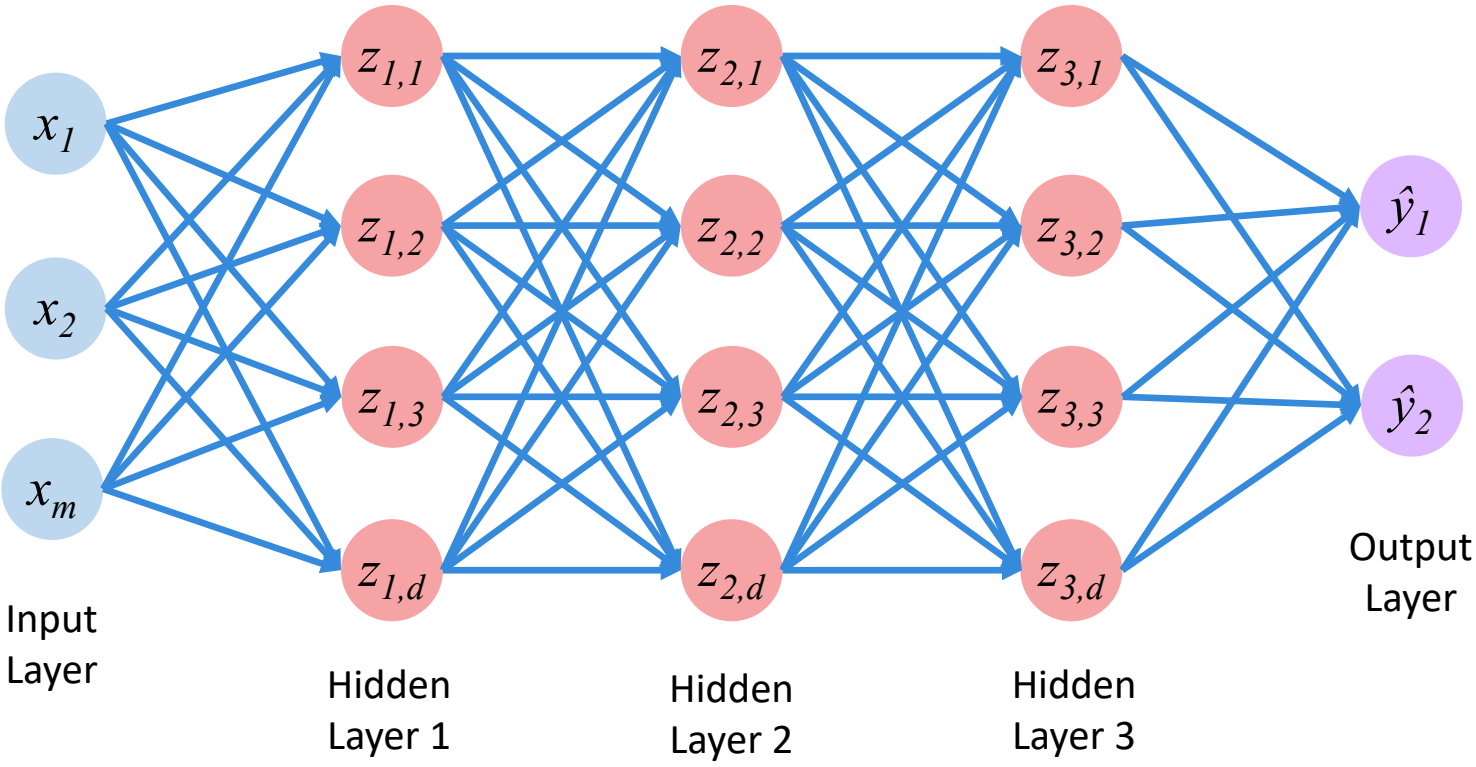
- Feature selection by assigning insignificant input features with zero weight and useful features with a non-zero weight.
- λ is the penalty term or regularization parameter which determines how much to penalizes the weights.
  - When λ is zero then the regularization term becomes zero. We are back to the original Loss function.
  - When λ is large, we penalizes the weights and they become close to zero. This results is a very simple model having a high bias or is underfitting.
- w_i are the weights

Types of regularisation

- L2 regularisation (Ridge)
  - Forces the weights to be small but does not make them zero (non sparse solution)
  - Not robust to outliers as square terms blows up the error differences of the outliers and the regularization term tries to fix it by penalizing the weights.
  - Performs better when all the input features influence the output and all with weights are of roughly equal size

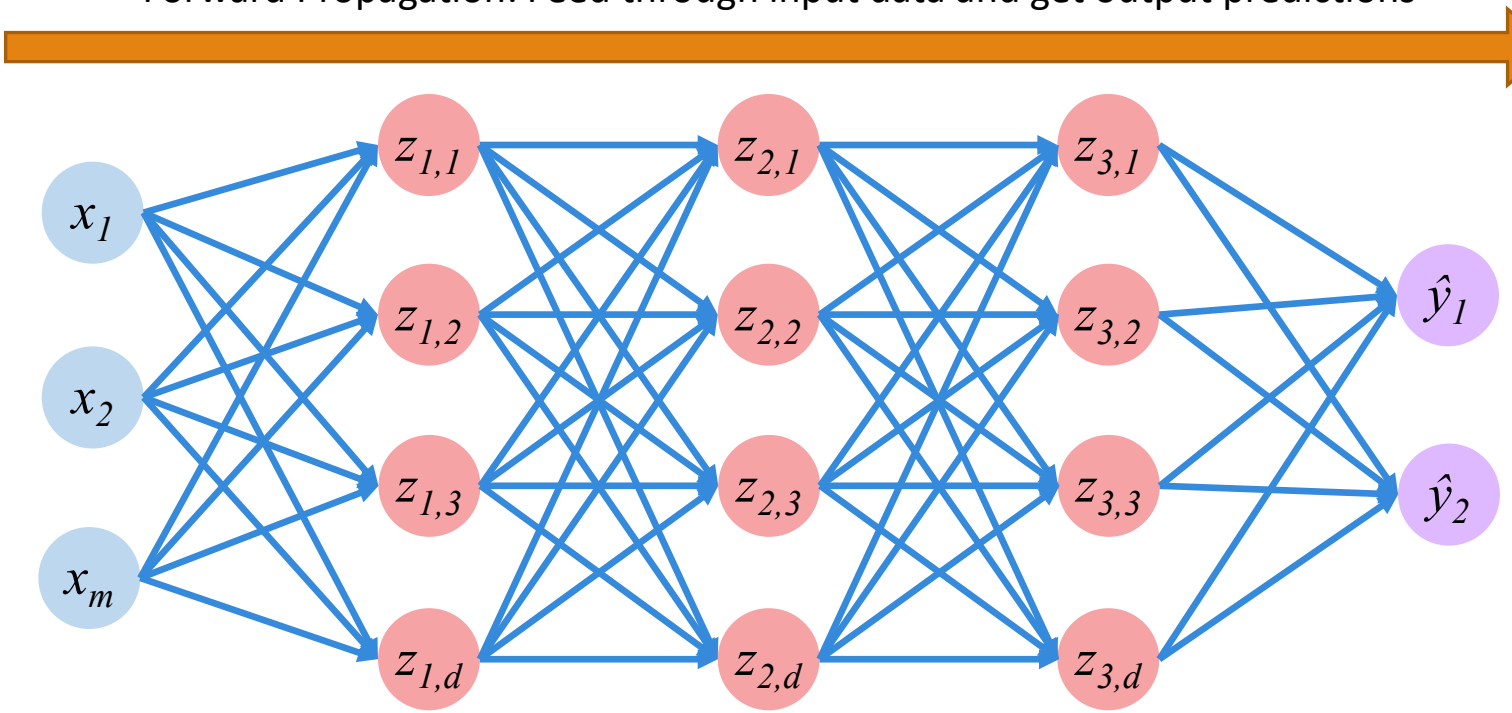
$$L(x, y) \equiv \sum_{i=1}^{n} (y_i - f(x_i; w_i))^2 + \lambda \sum_{i=1}^{n} w_i^2$$

Tying Everything Together

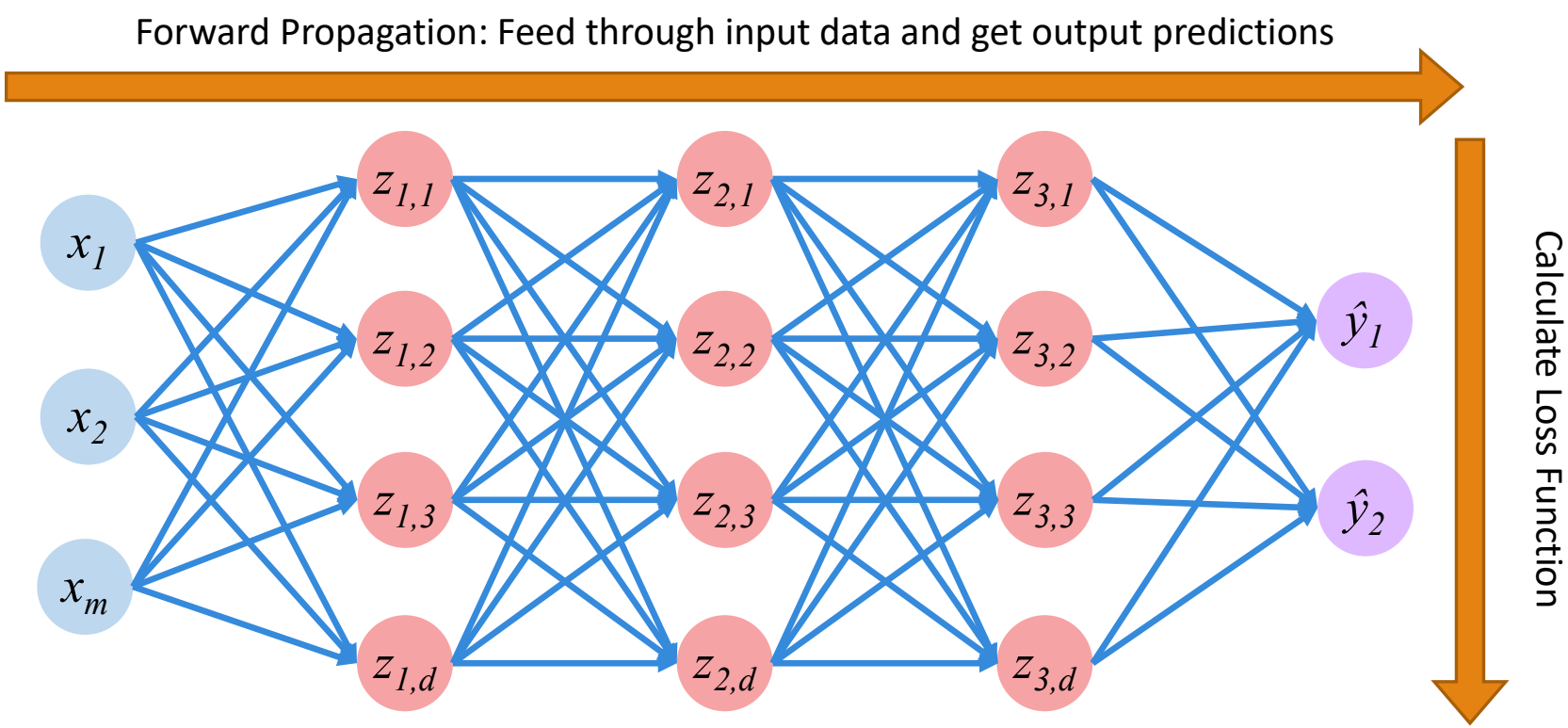


Tying Everything Together

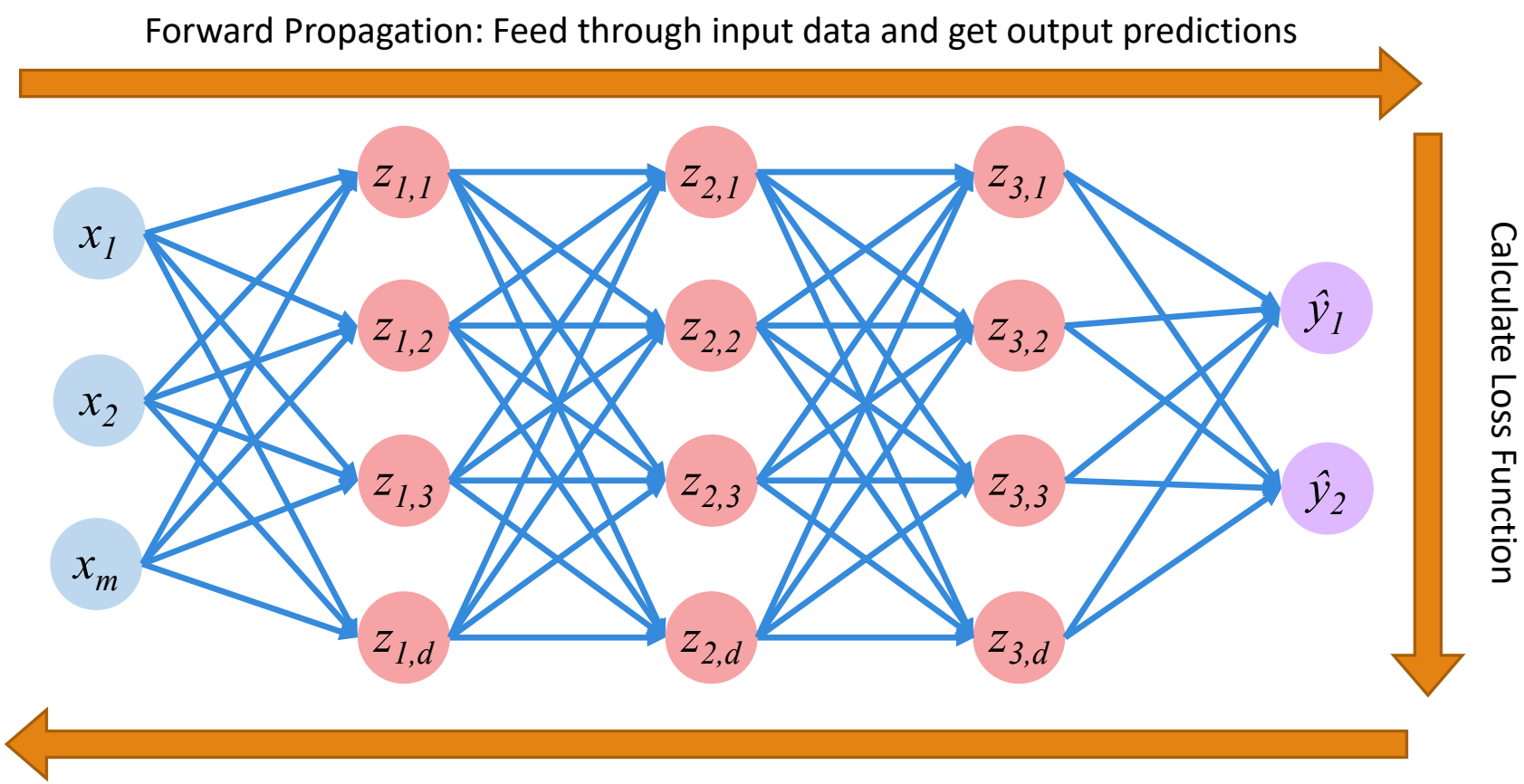
Forward Propagation: Feed through input data and get output predictions



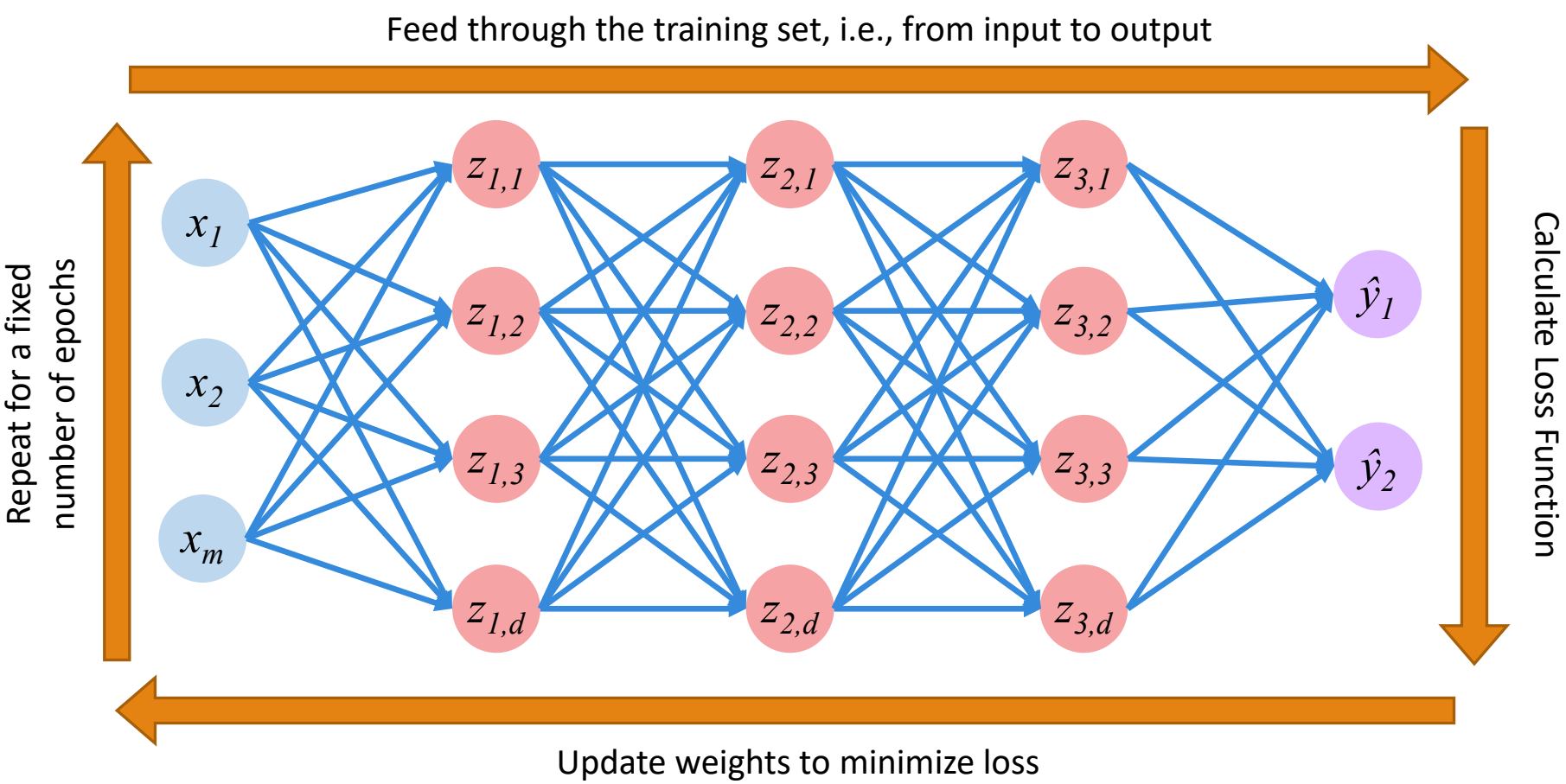
Tying Everything Together



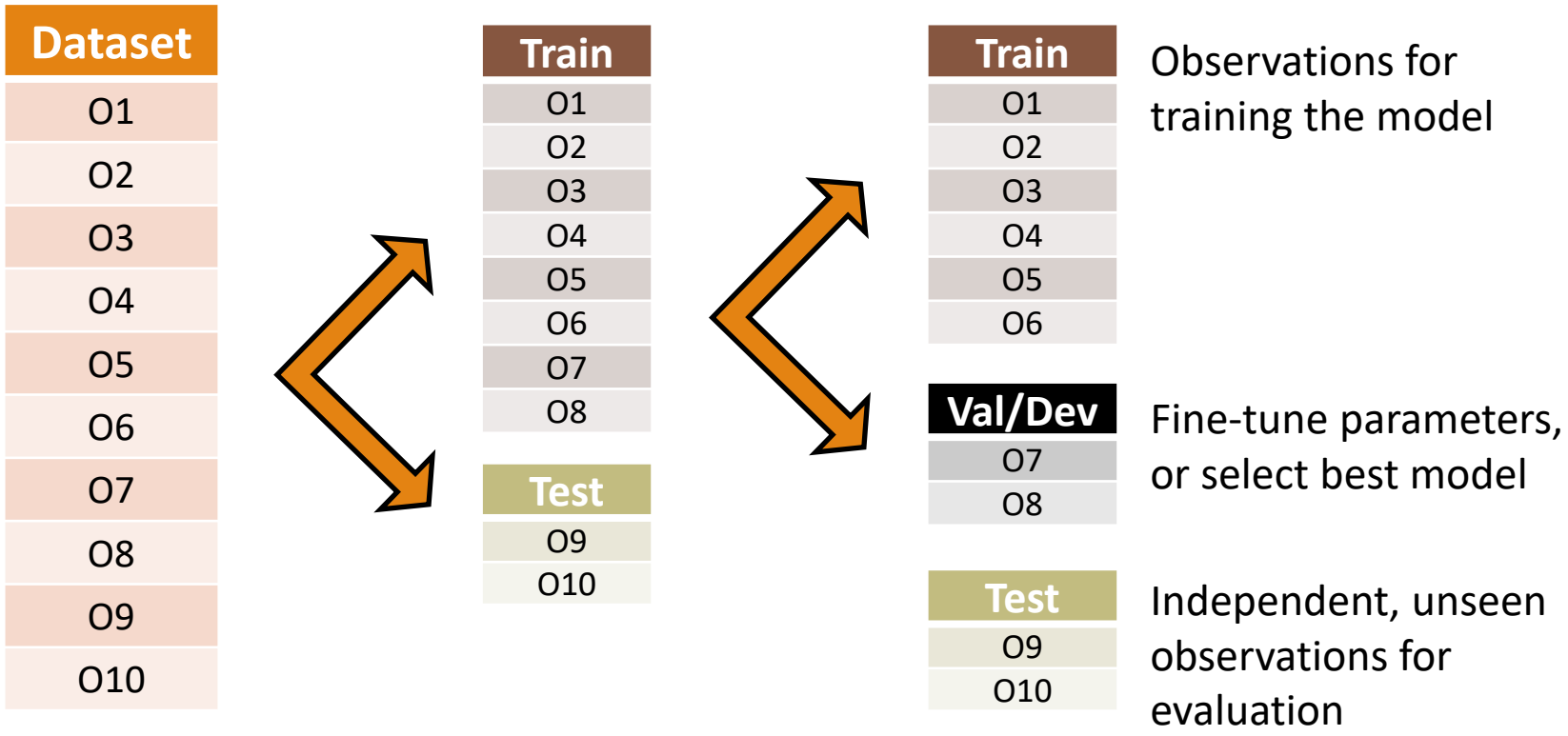
Tying Everything Together



Tying Everything Together



Dataset Split



Dataset Split

- Random Resampling
  - Random sampling into train/validation/test sets
- Stratified Sampling
  - Same as above but maintain class/categorical distribution
- Time-based Sampling
  - Split into train/validation/test sets based on time
- K-Fold Cross Validation
  - Split into K equal-sized partitions, iteratively test on one partition while training on the remaining K-1 partitions
- Leave-One-Out Validation
  - Same as above, with K=1

Summary

- Understand the concept of a perceptron and how multiple perceptrons work in the context of neural networks
- Understand how to train and use a neural network, including the roles that activation functions, losses and optimizers play
- Be aware of techniques to prevent overfitting in neural networks
- Be able to use a simple neural network to perform a certain task, e.g., compute the output for a prediction

Reminder: Quiz 1 (Week 5)

- Date/Time:
  - Week 5 Lecture Session, Tue 20 Feb 2024, 3pm
- Venue:
  - LT5 (our usual lecture venue)
- Topics covered:
  - Weeks 1 to 4
- Format:
  - MCQs and open-ended questions. Completed within 60min
  - 1 x A4 cheatsheet (double-sided) allowed, calculators allowed. Cheatsheet can be printed or handwritten.
  - 1 x A4 cheatsheet (double-sided) allowed. Can be printed or hand-written
  - You will not be asked to produce codes but may be given partial pseudo codes and asked to do something with it, e.g., complete, explain, correct, etc.

Guest Lecture (Week 6 / Compulsory)

- Date/Time: Tue, 27 Feb 2024, 3pm to 5+pm (Refreshments provided)
- Venue: SUTD Albert Hong Lecture Theatre 1 (Building 1 Level 1)
- Speaker: James Chong, Lead, The Tesseract, NCS Group
- Topic: “How to Sell a Better Mousetrap” – How Advanced Analytics and Generative AI is changing the world and how to convince people to use it
- Short Abstract: rom machine learning to large language models, advances in technologies like Advanced Analytics and Generative AI has upended the world, starting with being able to make sense and predict how things will run, to understanding the written word and conversations, to helping write better IT code. But how do you get people to accept and pay for the solutions based on these ideas?

Credits

- Images thanks to:
  - MIT 6.S191
  - https://towardsdatascience.com/https-medium-com-piotr-skalski92-deep-dive-into-deep-networks-math-17660bc376ba
  - https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/
  - https://colah.github.io/posts/2015-08-Understanding-LSTMs/
  - https://colah.github.io/