# Introduction to Deep Learning (I2DL)

## Exercise 6: Hyperparameter Tuning

# Today's Outline

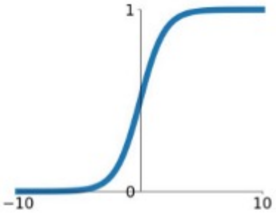
- Review Solution Ex5: Sigmoid Activation function
- Exercise 6: Hyperparameter Tuning

# Activation functions

# Activation functions

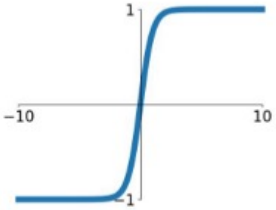
**Sigmoid**

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



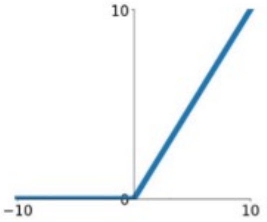

**tanh**

$$\tanh(x)$$



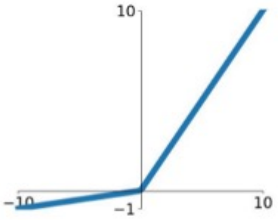

**ReLU**

$$\max(0, x)$$




**Leaky ReLU**

$$\max(0.1x, x)$$

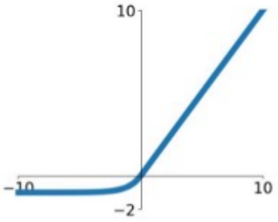



**Maxout**

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

**ELU**

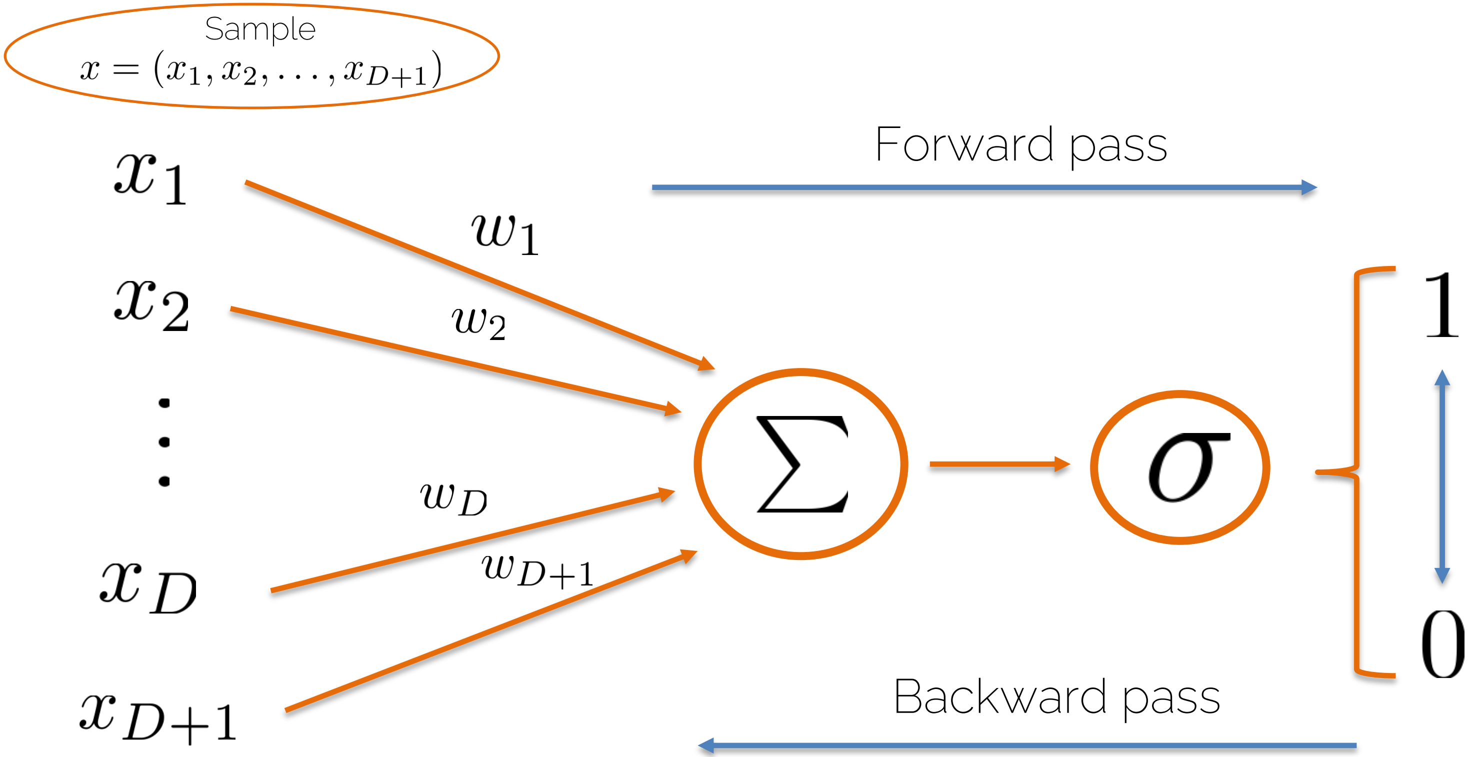
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$






Source: https://miro.medium.com/max/2000/1*4ZEDRpFuCIpUjNgjDdT2Lg.png

# Activation function: Sigmoid



# Sigmoid: Forward pass

- Definition of the Sigmoid function:

$$\sigma : \mathbb{R} \rightarrow \mathbb{R}, \sigma(x) = \frac{1}{1 + e^{-x}}$$

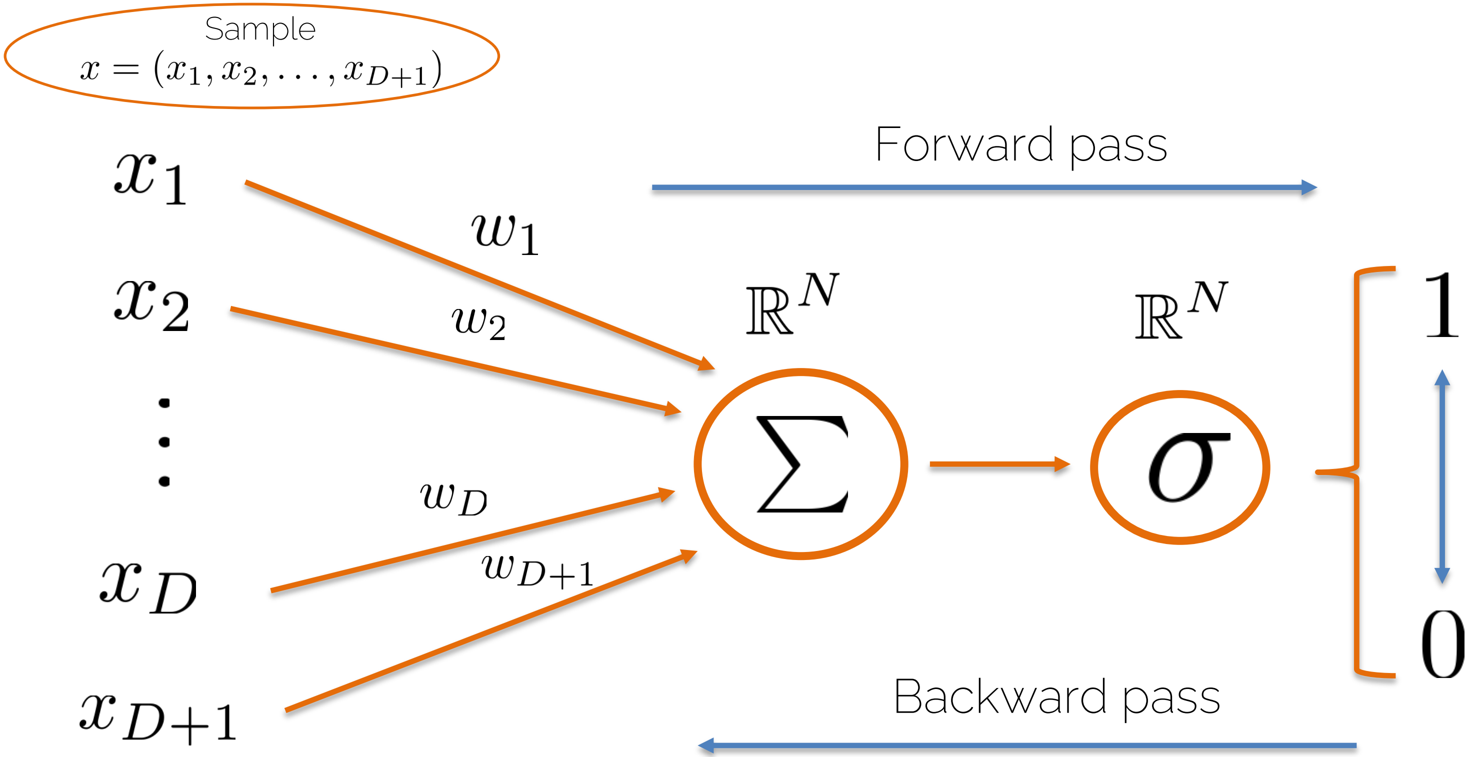
- Derivative of the sigmoid function:

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) \cdot (1 - \sigma(x))$$

- Application of the Sigmoid function in higher dimension:

$$\tilde{\sigma} : \mathbb{R}^N \rightarrow \mathbb{R}^N, \tilde{\sigma}(x) = \begin{pmatrix} \sigma(x_1) \\ \sigma(x_2) \\ \vdots \\ \sigma(x_N) \end{pmatrix}$$

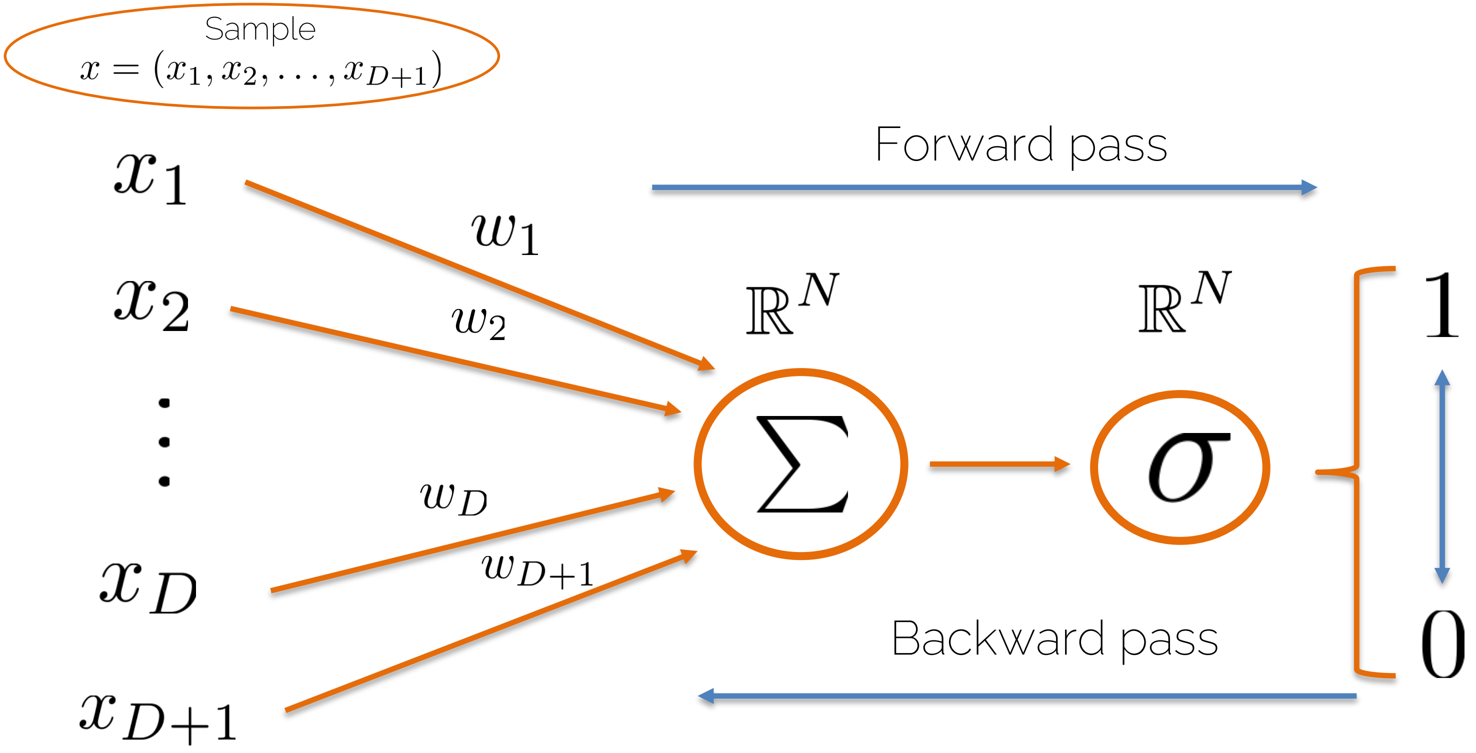
# Activation function: Sigmoid



# Sigmoid: Forward pass

```
def forward(self, x):
    """
    :param x: Inputs, of any shape

    :return out: Output, of the same shape as x
    :return cache: Cache, for backward computation, of the same shape as x
    """
    shape = x.shape
    outputs, cache= np.zeros(shape), np.zeros(shape)
    ########################################################################
    # TODO:                                                                #
    # Implement the forward pass of Sigmoid activation function            #
    ########################################################################
    outputs = 1 / (1 + np.exp(-x))
    cache = outputs
    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
    return outputs, cache
```

# Activation function: Sigmoid



# Sigmoid: Backward Pass

- The derivative of the sigmoid function is thus given a N x N - sized Jacobian matrix.

$$\tilde{\sigma} : \mathbb{R}^N \to \mathbb{R}^N, \tilde{\sigma}(x) = \begin{pmatrix} \sigma(x_1) \\ \sigma(x_2) \\ \vdots \\ \sigma(x_N) \end{pmatrix}$$

$$J_\sigma : \mathbb{R}^N \to \mathbb{R}^{N \times N}, J_\sigma = \begin{pmatrix} \frac{\partial \sigma(x_1)}{\partial x_1} & \frac{\partial \sigma(x_1)}{\partial x_2} & \cdots & \frac{\partial \sigma(x_1)}{\partial x_N} \\ \frac{\partial \sigma(x_2)}{\partial x_1} & \frac{\partial \sigma(x_2)}{\partial x_2} & \cdots & \frac{\partial \sigma(x_2)}{\partial x_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \sigma(x_N)}{\partial x_1} & \frac{\partial \sigma(x_N)}{\partial x_2} & \cdots & \frac{\partial \sigma(x_N)}{\partial x_N} \end{pmatrix} = \begin{pmatrix} \frac{\partial \sigma(x_1)}{\partial x_1} & 0 & \cdots & 0 \\ 0 & \frac{\partial \sigma(x_2)}{\partial x_2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{\partial \sigma(x_N)}{\partial x_N} \end{pmatrix}$$

# Sigmoid: Backward pass

$$J_\sigma = \begin{pmatrix} \frac{\partial\sigma(x_1)}{\partial x_1} & 0 & \dots & 0 \\ 0 & \frac{\partial\sigma(x_2)}{\partial x_2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{\partial\sigma(x_N)}{\partial x_N} \end{pmatrix}$$

```
def backward(self, dout, cache):
    """
    :return: dx: the gradient w.r.t. input X, of the same shape as X
    """
    dx = None
    ########################################################################
    # TODO:                                                                #
    # Implement the backward pass of Sigmoid activation function           #
    ########################################################################
    dx = dout * cache * (1 - cache)
    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
    return dx
```

$$J_\sigma = \begin{pmatrix} \frac{\partial\sigma(x_1)}{\partial x_1} \\ \frac{\partial\sigma(x_2)}{\partial x_2} \\ \vdots \\ \frac{\partial\sigma(x_N)}{\partial x_N} \end{pmatrix}$$

On paper

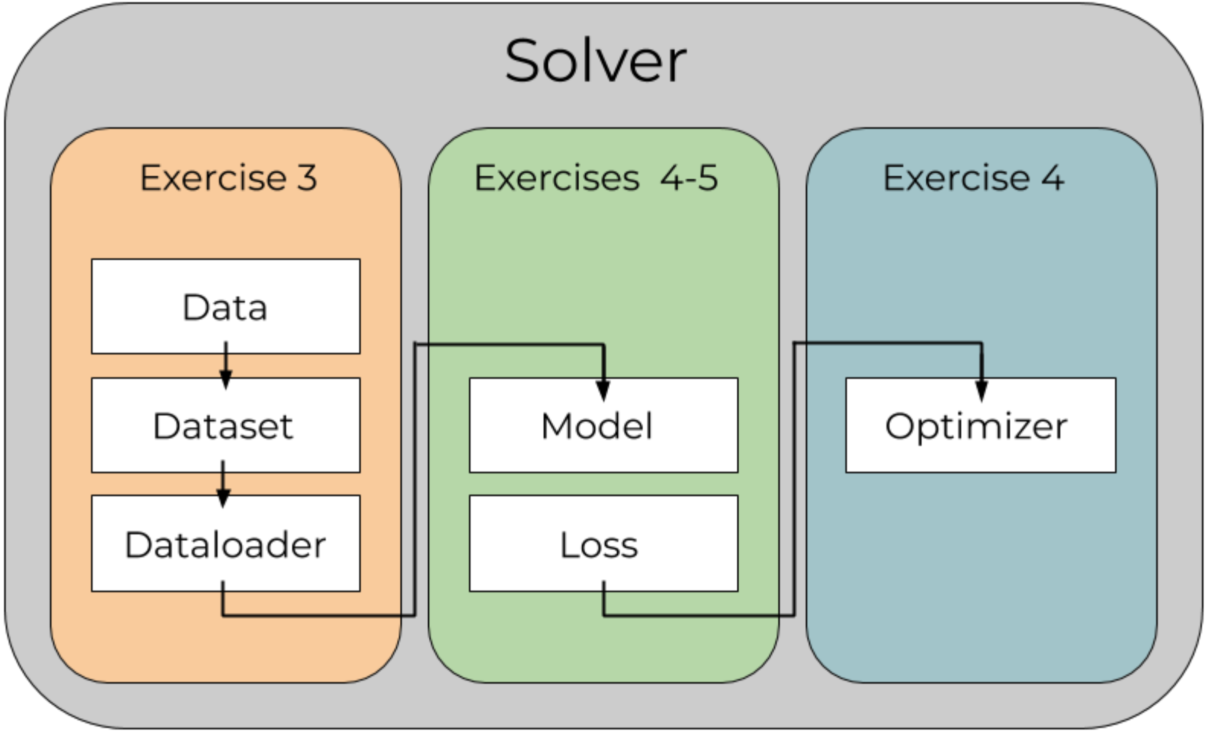
- Cache is an N x 1 vector
- Derivative of Sigmoid is N x N matrix
- Multiplication is normal matrix multiplication
- Dout is the upward gradient with dimension N x M (for M a natural number)

Numpy arrays

- Cache is a N x 1 vector
- Derivative of Sigmoid is given as N x 1 vector
- Multiplication: Numpy.multiply() which is componentwise multiplication
- Dout is upward gradient with dimension N x M (for M a natural number)

# Exercise 6: Hyperparameter Tuning

# Recap: Pillars of Deep Learning



# Goal of exercise 6

- Use existing implementations
  - Reworked implementations of previous exercises
  - We will provide you with additional implementations of all required tools to run sample methods proposed in the lecture


- Learn about neural network debugging strategies and hyperparameter search

# Previously: Dataset

```python
class ImageFolderDataset(Dataset):
    """CIFAR-10 dataset class"""
    def __init__(self, transform=None, mode='train',
        limit_files=None,
        split={'train': 0.6, 'val': 0.2, 'test': 0.2},
        *args, **kwargs): ...

    @staticmethod
    def _find_classes(directory): ...

    def select_split(self, images, labels, mode): ...

    def make_dataset(self, directory, class_to_idx, mode): ...

    def __len__(self): ...

    @staticmethod
    def load_image_as_numpy(image_path): ...

    def __getitem__(self, index): ...
```

```python
# Create a train, validation and test dataset.
datasets = {}
for mode in ['train', 'val', 'test']:
    crt_dataset = ImageFolderDataset(
        mode=mode,
        root=cifar_root,
        download_url=download_url,
        transform=compose_transform,
        split={'train': 0.6, 'val': 0.2, 'test': 0.2}
    )
    datasets[mode] = crt_dataset
```

# Previously: Data Loader

```python
class DataLoader:
    """
    Dataloader Class
    Defines an iterable batch-sampler over a given dataset
    """
    def __init__(self,
        dataset,
        batch_size=1,
        shuffle=False,
        drop_last=False): ...

    def __iter__(self): ...

    def __len__(self): ...
```

```python
# Create a dataloader for each split.
dataloaders = {}
for mode in ['train', 'val', 'test']:
    crt_dataloader = DataLoader(
        dataset=datasets[mode],
        batch_size=256,
        shuffle=True,
        drop_last=True,
    )
    dataloaders[mode] = crt_dataloader
```

# Previously: Solver

```python
class Solver(object):
    """
    A Solver encapsulates all the logic necessary for training classification
    or regression models.
    The Solver performs gradient descent using the given learning rate.
    """

    def __init__(self, model, train_dataloader, val_dataloader,
        loss_func=CrossEntropyFromLogits(), learning_rate=1e-3,
        optimizer=Adam, verbose=True, print_every=1,
        lr_decay = 1.0, **kwargs): ...

    def _reset(self): ...

    def _step(self, X, y, validation=False): ...

    def train(self, epochs=100, patience = None): ...

    def get_dataset_accuracy(self, loader): ...

    def update_best_loss(self, val_loss, train_loss): ...
```

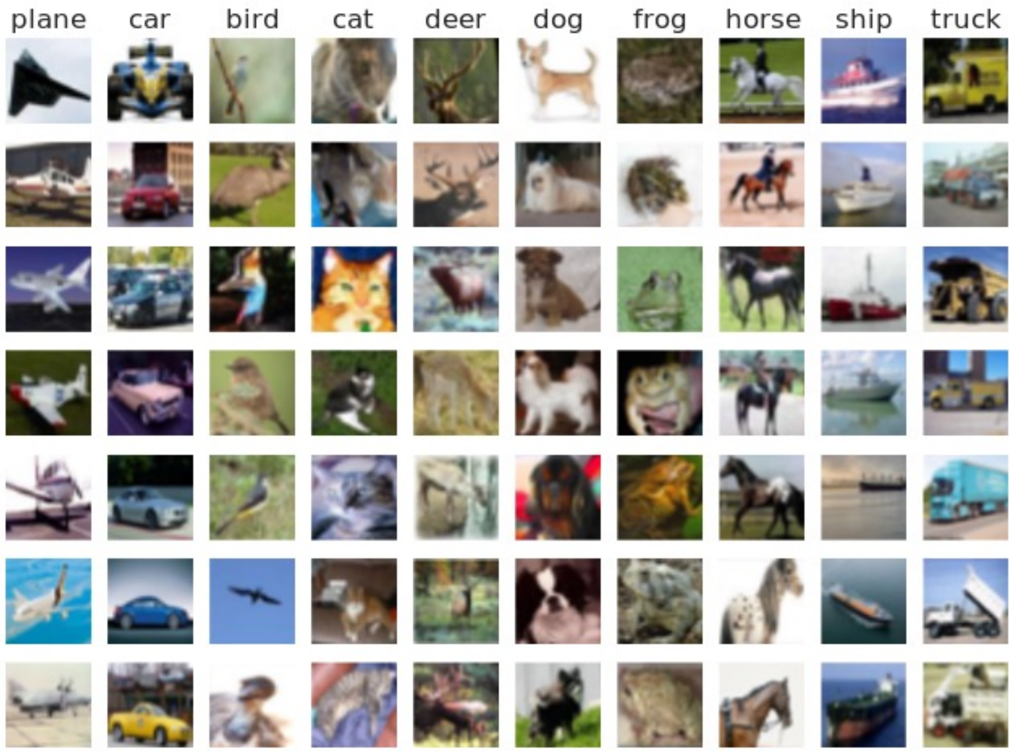
```python
solver = Solver(model,
                dataloaders['train'],
                dataloaders['val'],
                learning_rate=0.001,
                loss_func=MSE(),
                optimizer=SGD)

solver.train(epochs=epochs)
```

# Previously: Classification Network

```python
class ClassificationNet(Network):
    """
    A fully-connected classification neural network with configurable
    activation function, number of layers, number of classes, hidden size and
    regularization strength.
    """

    def __init__(self,
        activation=Sigmoid(), num_layer=2,
        input_size=3 * 32 * 32, hidden_size=100,
        std=1e-3, num_classes=10, reg=0, **kwargs): ...

    def forward(self, X): ...

    def backward(self, dy): ...

    def save_model(self): ...

    def get_dataset_prediction(self, loader): ...
```

```python
# Instantiate a new model.
model = ClassificationNet(activation=Sigmoid(),
                          num_layer=num_layer,
                          reg=reg,
                          num_classes=10)

# X is a batch of training features
# X.shape = (batch_size, features_size)
y_out = model.forward(X)

# dout is the gradient of the loss function
#   w.r.t the output of the network.
# dout.shape = (batch_size, )
model.backward(dout)
```

# Submission Goal: Cifar10 Classification



# Previously: Binary Cross Entropy Loss

$$BCE\left(\hat{y}, y\right) = \frac{1}{N}\sum_{i=1}^{N}\left[-y_i \log\left(\hat{y}_i\right) - (1 - y_i)\log(1 - \hat{y}_i)\right]$$

Where

- N is the number of samples
- $\hat{y}_i$ is the network's prediction for sample i
- $y_i$ is the ground truth label (0 or 1)

# New: Multiclass Cross Entropy Loss

$$CE(\hat{y}, y) = \frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{C} \left[ -y_{ik} \log(\hat{y}_{ik}) \right]$$

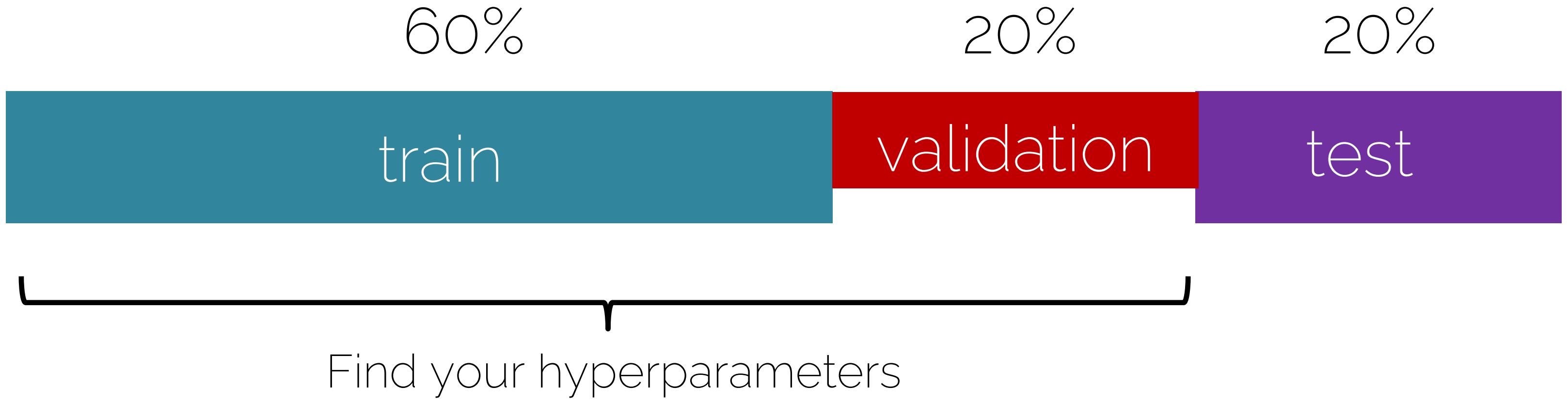
We implemented this for you! More on this topic in the next lecture.

Where

- N is the number of samples
- $\hat{y}_{ik}$ is the network's predicted probability for the kth class when given the sample i
- $y_{ik}$ is the ground truth label which is either 1 if the ith sample is of class k or zero otherwise

# Basic Recipe for Machine Learning

- Split your data



# Basic Recipe for Machine Learning

- Split your data

| 60% | 20% | 20% |
|---|---|---|
| train | validation | test |

Example scenario

Ground truth error ...... 1%

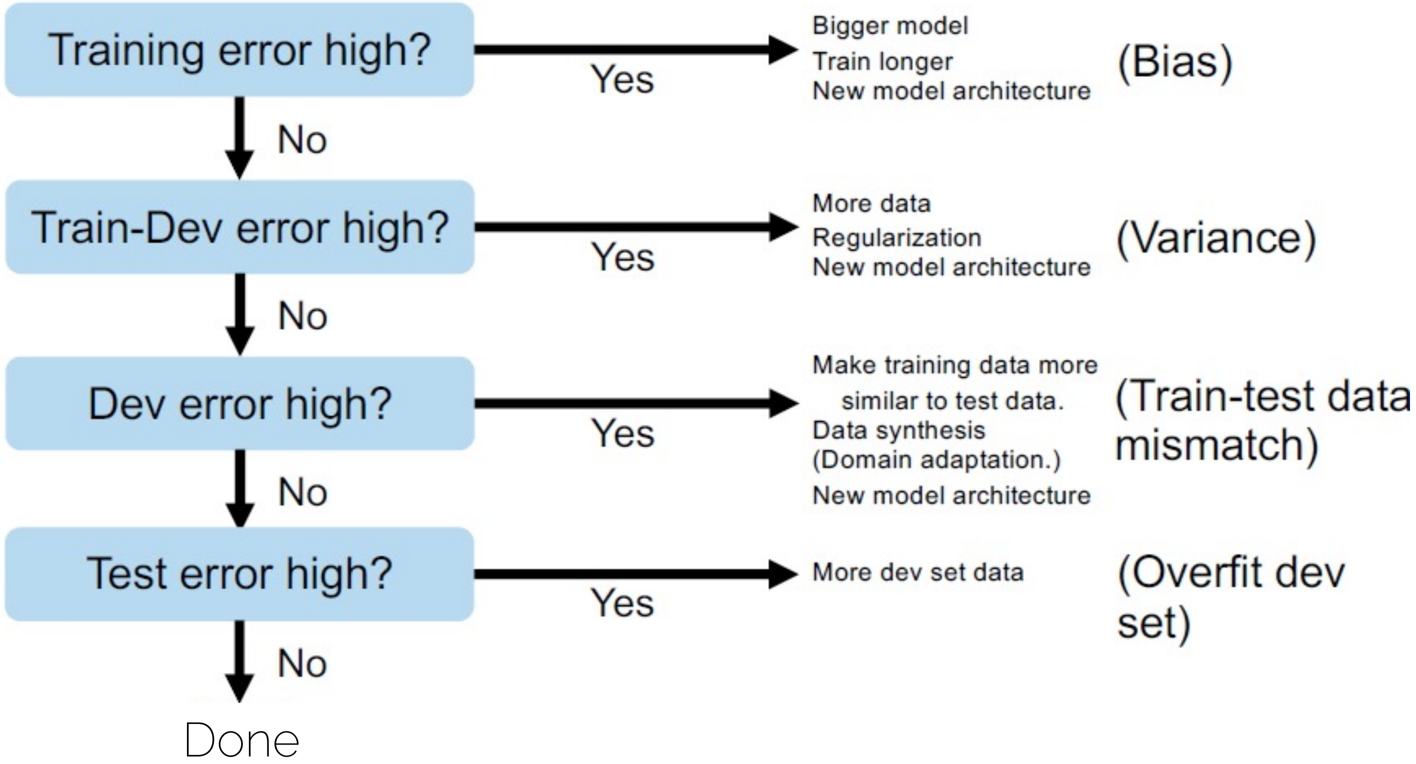
Training set error ....... 5%

Val/test set error ....... 8%

Bias (underfitting)

Variance (overfitting)

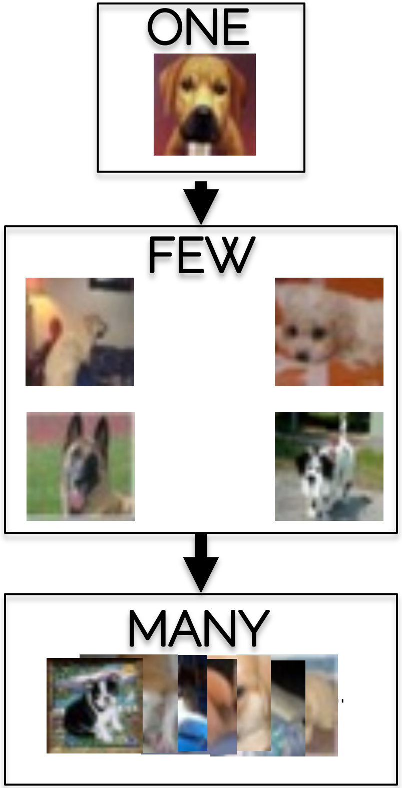
# Basic Recipe for Machine Learning




Credits: A. Ng

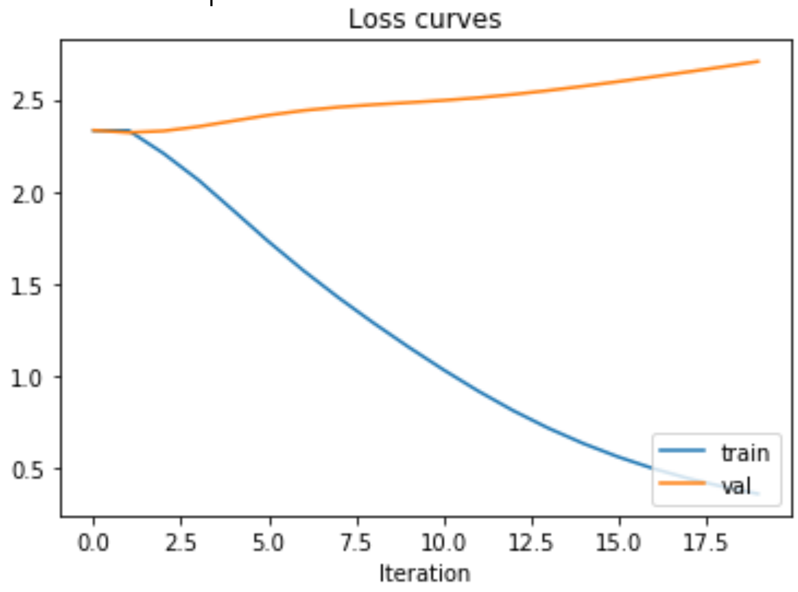
# How to Start

- Start with single training sample
  - Check if output correct
  - Overfit → train accuracy should be 100% because input just memorized
- Increase to handful of samples
- Move from overfitting to more samples
  - At some point, you should see generalization

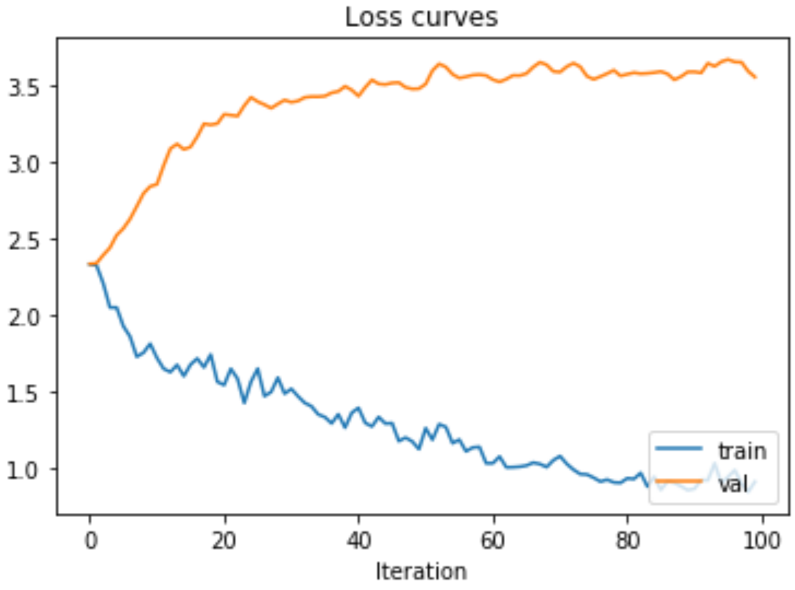


# How to Start

- Overfit a single training sample



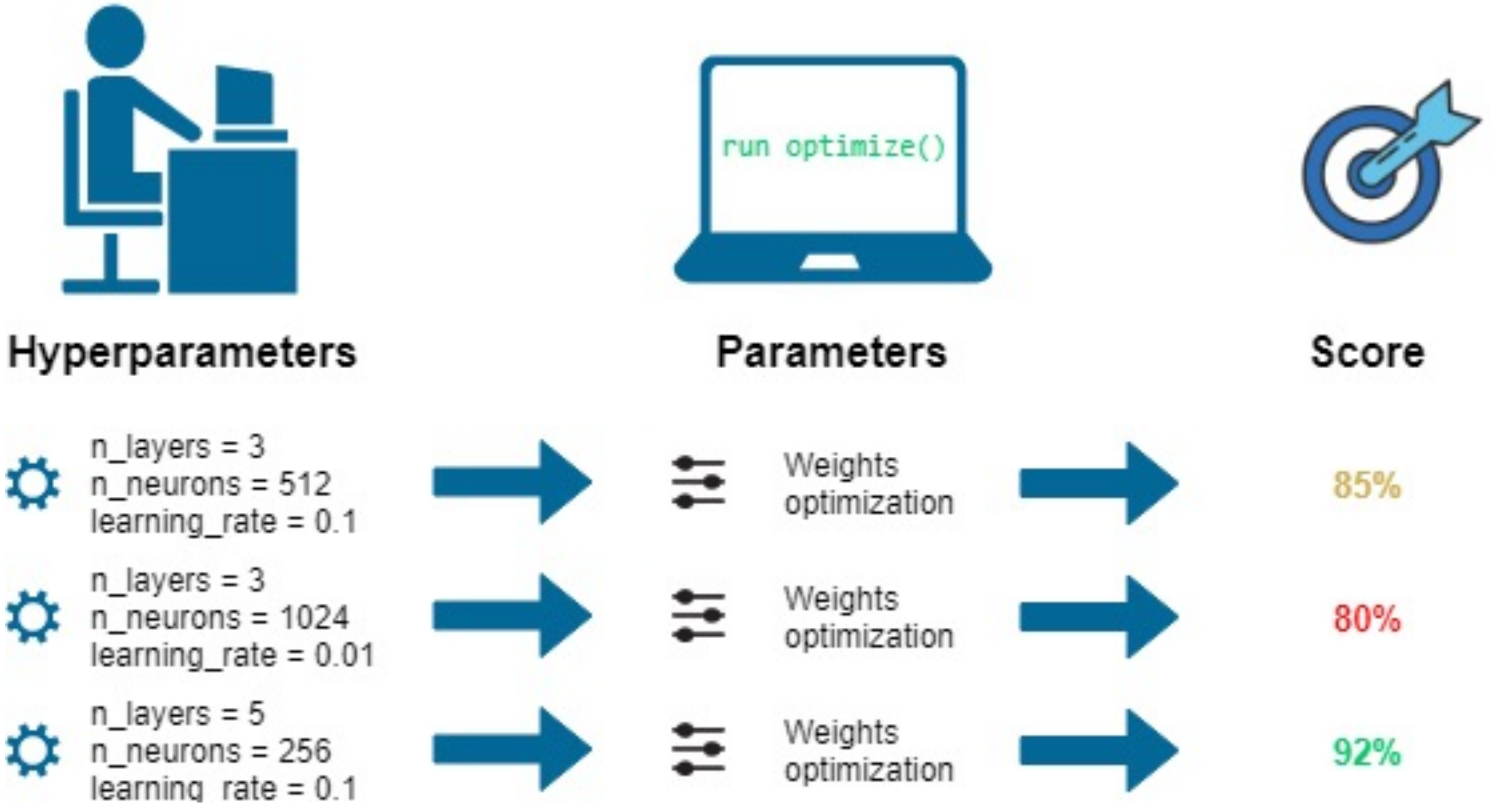

- Then a few samples



# Hyperparameters

- Network architecture (e.g., num layers, #weights)
- Number of iterations
- Learning rate(s) (i.e., solver parameters, decay, etc.)
- Regularization (more later next lecture)
- Batch size
- …

# Hyperparameter Tuning




Source: https://images.deepai.org/glossary-terms/05c646fe1676490aa0b8cab0732a02b2/hyperparams.png

# How to find good Hyperparameters?

- Manual Search (trial and error)
- Automated Search:
  - Grid Search
  - Random Search

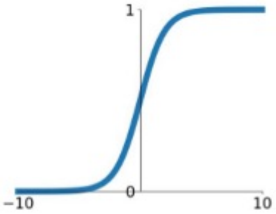
```python
from exercise_code.hyperparameter_tuning import grid_search

best_model, results = grid_search(
    dataloaders['train_small'], dataloaders['val_500files'],
    grid_search_spaces = {
        "learning_rate": [1e-2, 1e-3, 1e-4, 1e-5, 1e-6],
        "reg": [1e-4, 1e-5, 1e-6]
    },
    epochs=10, patience=5,
    model_class=ClassificationNet)
```

- Think about how different hyper parameters affect the model
  - E.g. Overfitting? -> Increase Regularization Strength, decrease model Capacity

# Optional: Activation Functions

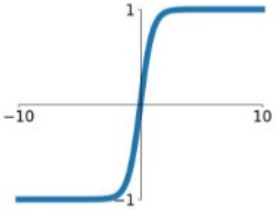
**Sigmoid**

$\sigma(x) = \frac{1}{1+e^{-x}}$



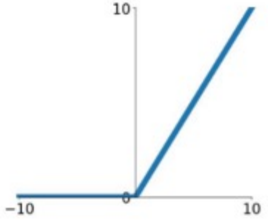

**tanh**

$\tanh(x)$



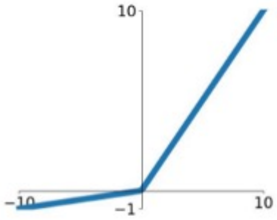

**ReLU**

$\max(0, x)$




**Leaky ReLU**

$\max(0.1x, x)$

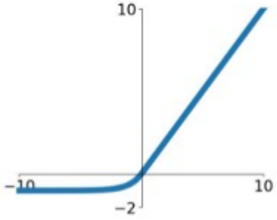



**Maxout**

$\max(w_1^T x + b_1, w_2^T x + b_2)$

**ELU**

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






Source: https://miro.medium.com/max/2000/1*4ZEDRpFuCIpUjNgjDdT2Lg.png

# Submission

- Your model's accuracy is all that counts!
  - At least 48% to pass the submission
  - There will be a leaderboard of all students!

**Leaderboard: Submission 6**

| Rank | User | Score | Pass |
|---|---|---|---|
| #1 | s0270 | 51.65 | ✔ |
| #2 | s0262 | 42.98 | ✘ |
| #3 | s0265 | 10.35 | ✘ |

# Exercise plan: Recap and Outlook

| Exercises | Framework |
| --- | --- |
| Exercise 03: Dataset and Dataloader<br>Exercise 04: Solver and Linear Regression<br>Exercise 05: Neural Networks<br>Exercise 06: Hyperparameter Tuning | Numpy<br>(Reinvent the wheel) |
| Exercise 07: Introduction to Pytorch<br>Exercise 08: MNIST with Pytorch | Pytorch/Tensorboard |
| Exercise 09: Convolutional Neural Networks<br>Exercise 10: Semantic Segmentation<br>Exercise 11: Recurrent Neural Networks | Applications<br>(Hands-off) |

# Summary

- Monday 29.11 : Watch Lecture 7
  - Training NN’s 2
- Wednesday 01.12 15:59: Submit exercise 6
- Thursday 02.12: Tutorial 7
  - PyTorch

Good luck &
see you next week ☺️