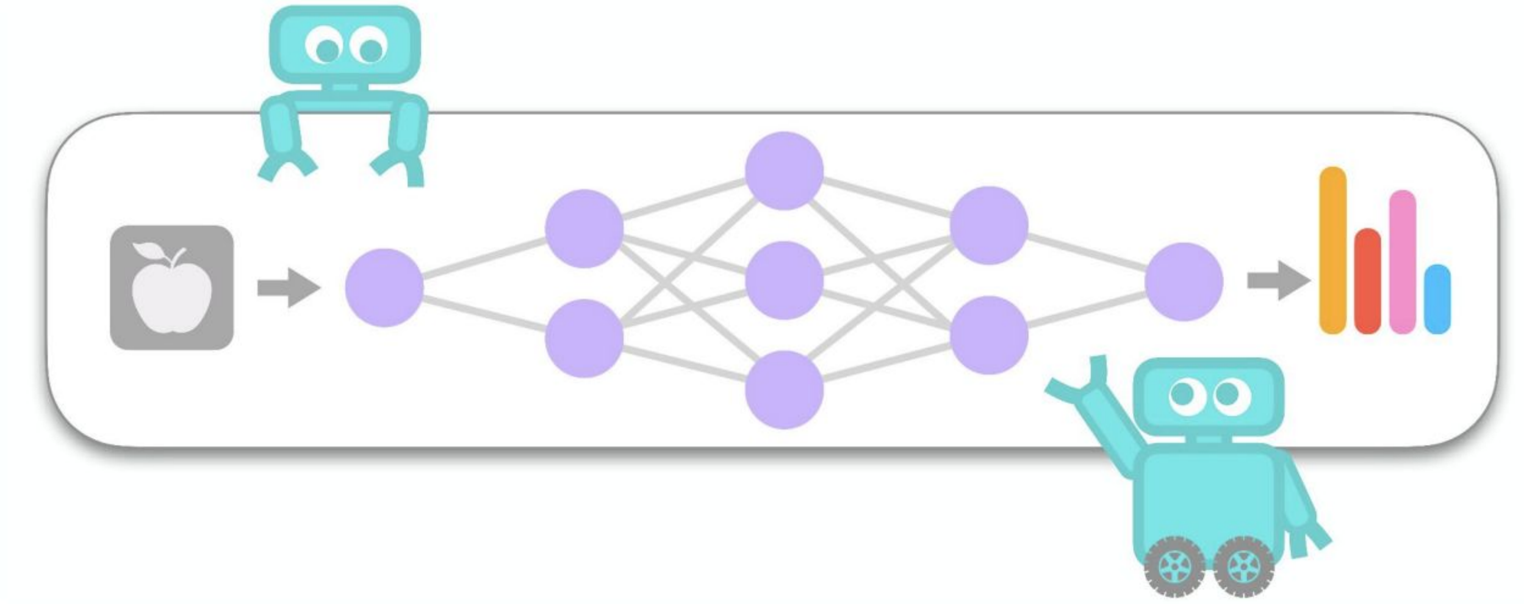


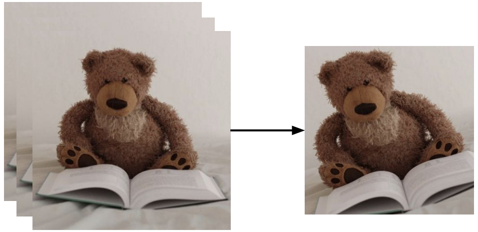
Training Tricks

Outline

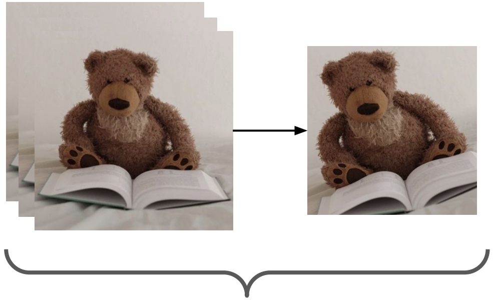
- Training a Neural Network (Review)
- Training Tricks
  - Data processing
  - Parameter tuning
  - Regularization
- Good practices

Training a Neural Network

Training a Neural Network

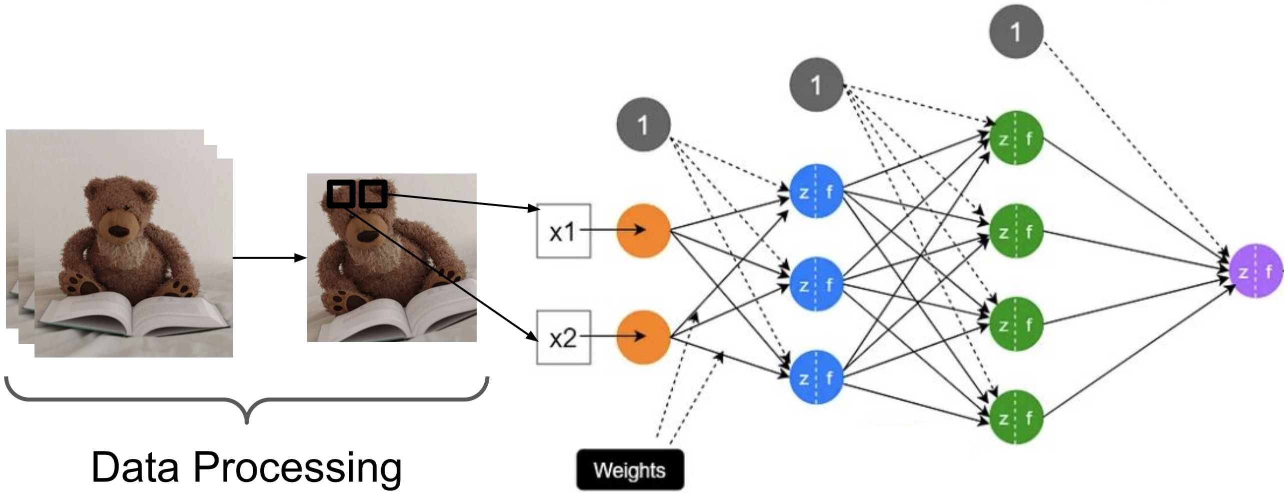


Training a Neural Network

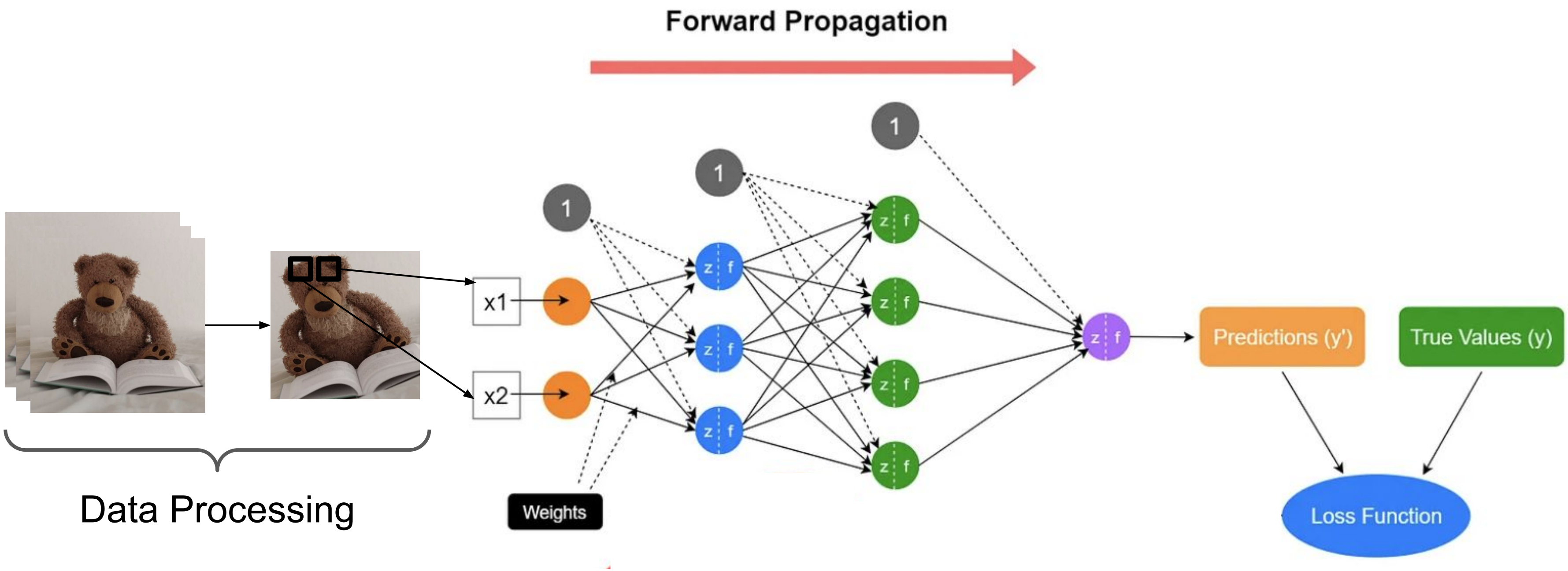


Data Processing

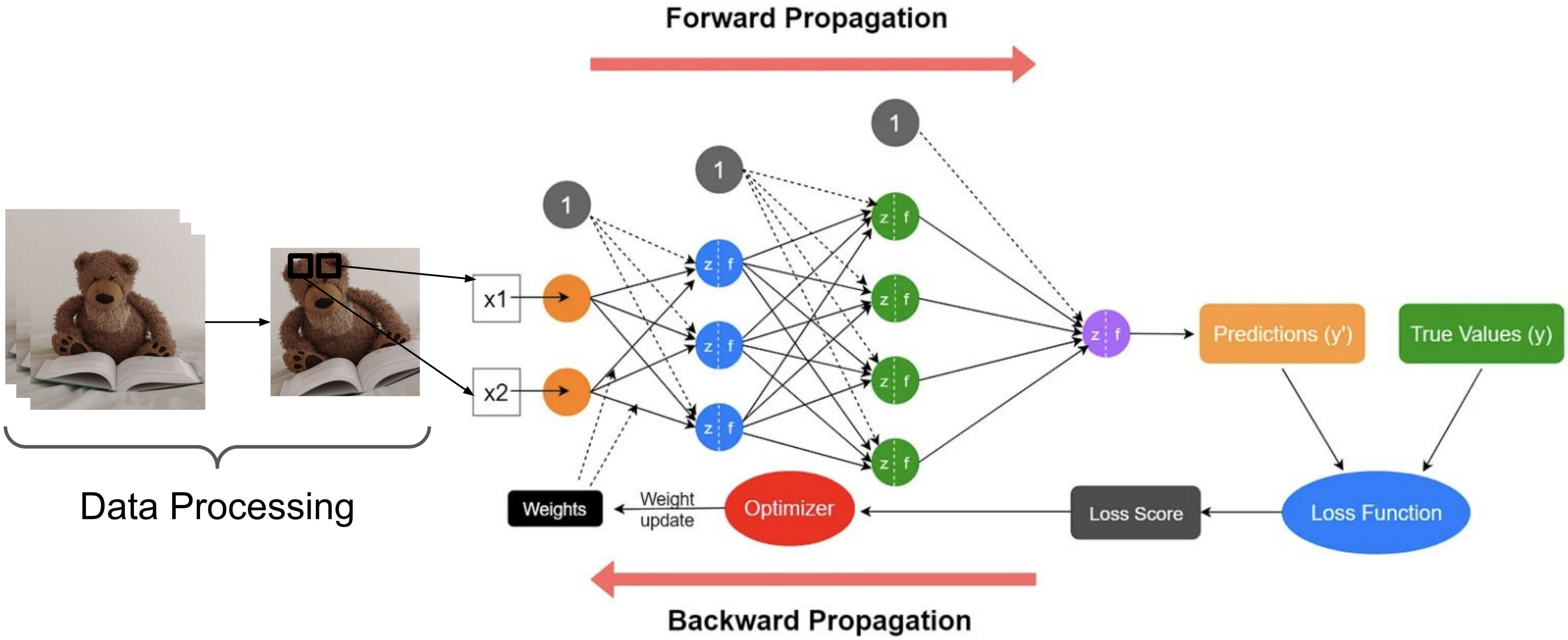
Training a Neural Network



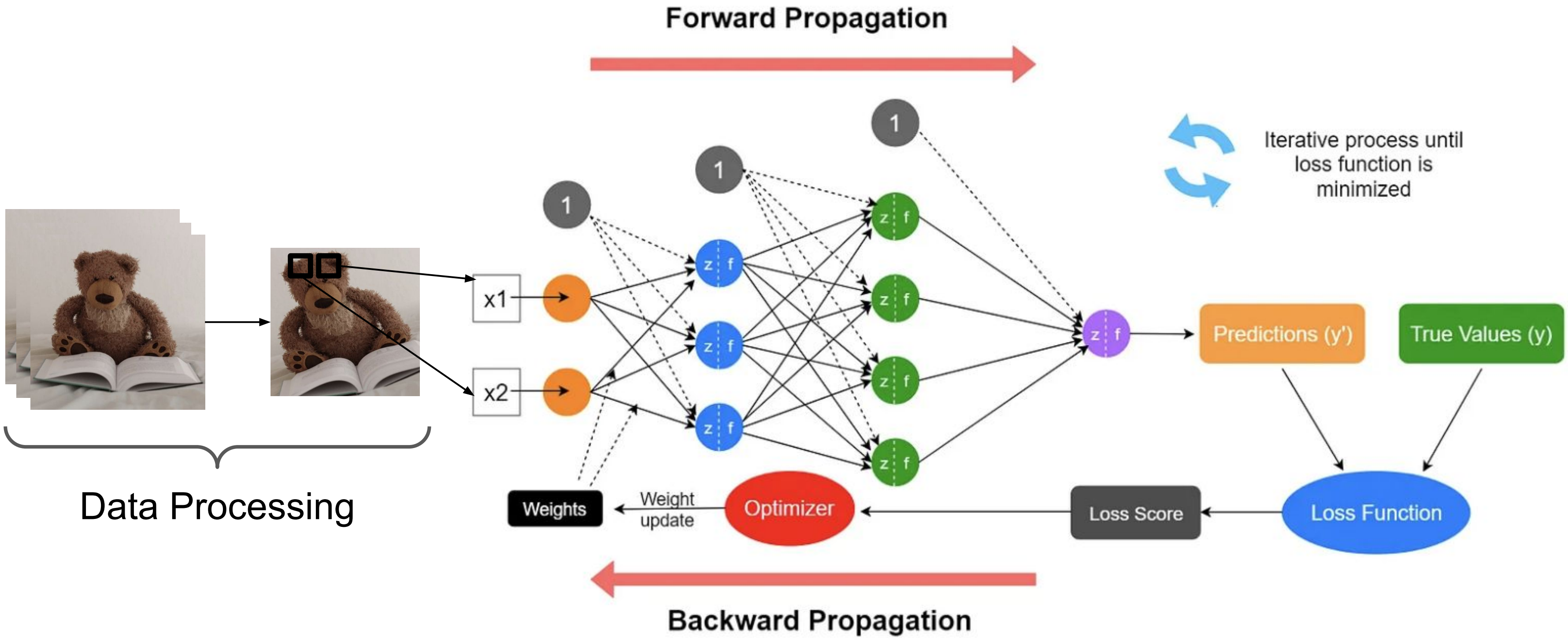
Training a Neural Network



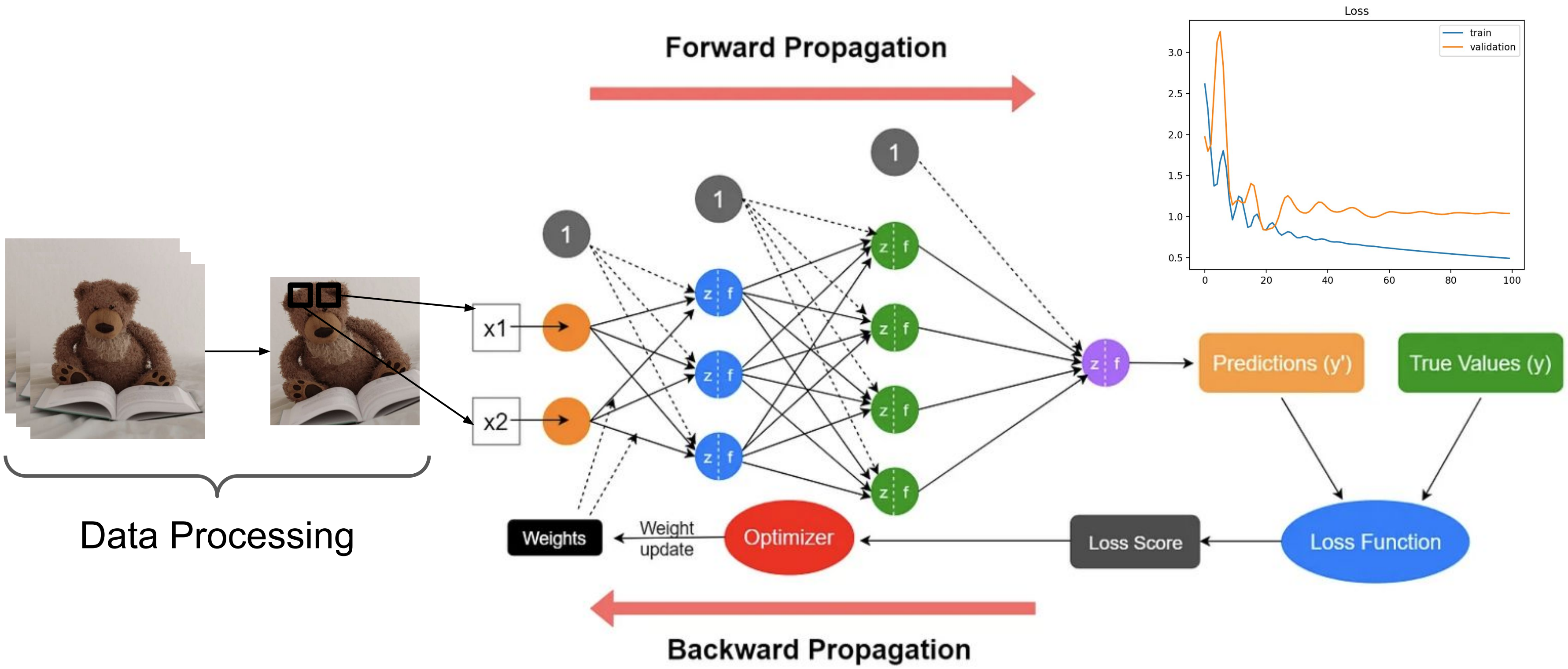
Training a Neural Network



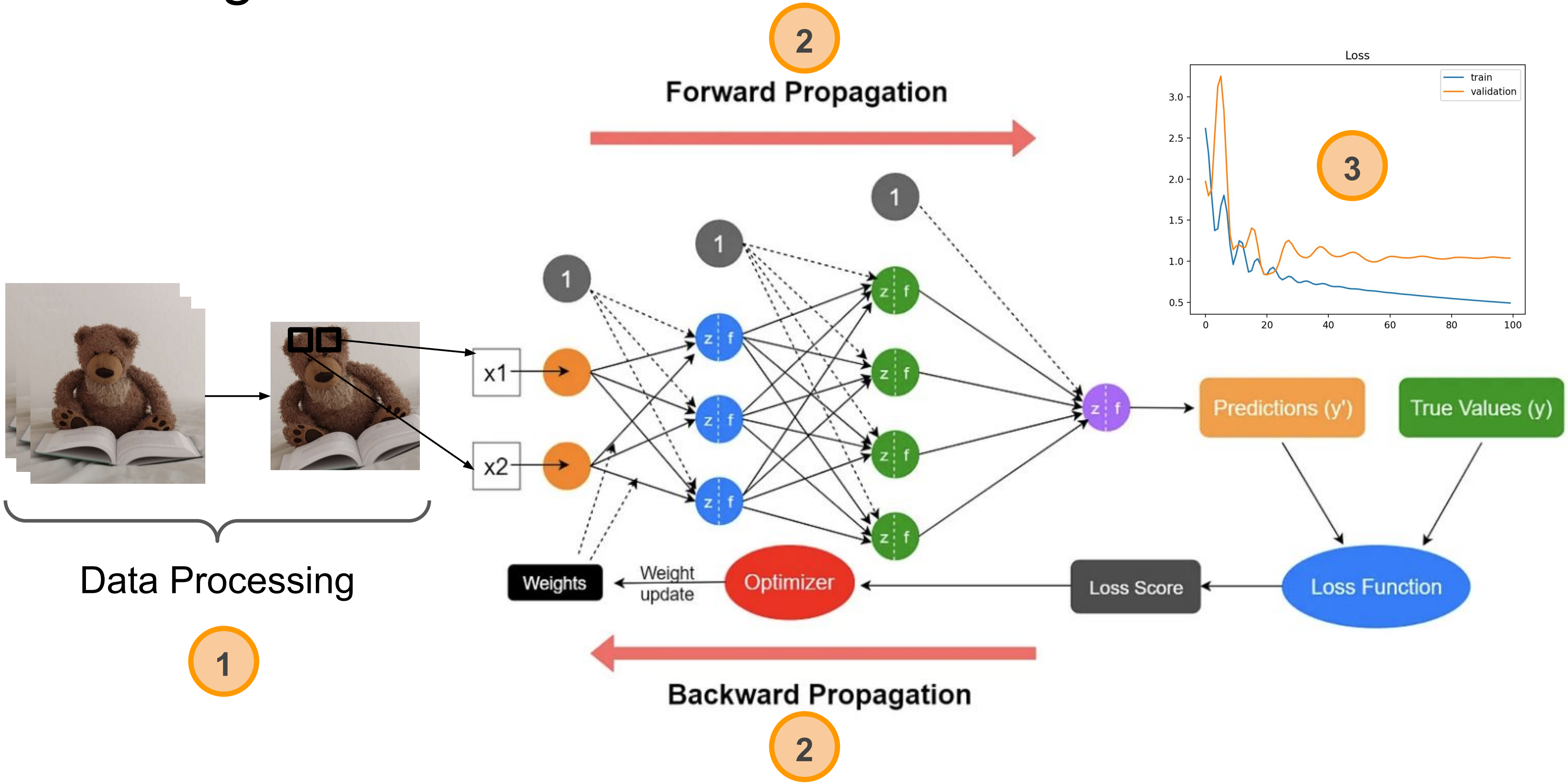
Training a Neural Network



Training a Neural Network



Training a Neural Network



Data Processing — Data Augmentation

Data Processing — Data Augmentation

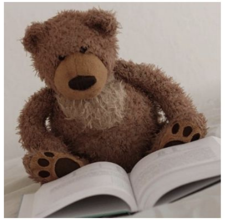
- Deep learning usually needs a lot of data to be properly trained.

Data Processing — Data Augmentation

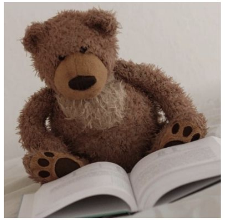
- Deep learning usually needs a lot of data to be properly trained.
- One easy way to get more data is to transform existing ones using data augmentation techniques.

1

Data Processing — Data Augmentation

Original	Flip	Rotation	Random crop
			
• Image without any modification	• Flipped with respect to an axis for which the meaning of the image is preserved	• Rotation with a slight angle • Simulates incorrect horizon calibration	• Random focus on one part of the image • Several random crops can be done in a row

Data Processing — Data Augmentation

Original	Flip	Rotation	Random crop
			
• Image without any modification	• Flipped with respect to an axis for which the meaning of the image is preserved	• Rotation with a slight angle • Simulates incorrect horizon calibration	• Random focus on one part of the image • Several random crops can be done in a row

Color shift	Noise addition	Information loss	Contrast change
			
• Nuances of RGB is slightly changed • Captures noise that can occur with light exposure	• Addition of noise • More tolerance to quality variation of inputs	• Parts of image ignored • Mimics potential loss of parts of image	• Luminosity changes • Controls difference in exposition due to time of day

Data Processing — How to Do It?

Data Processing — How to Do It?

This is what a typical transform pipeline could look like:

```
from torchvision.transforms import v2
transforms = v2.Compose([
    v2.ToImage(),  # Convert to tensor, only needed if you had a PIL image
    v2.ToDtype(torch.uint8, scale=True),  # optional, most input are already uint8 at this
point
    # ...
    v2.RandomResizedCrop(size=(224, 224), antialias=True),  # Or Resize(antialias=True)
    # ...
    v2.ToDtype(torch.float32, scale=True),  # Normalize expects float input
    v2.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
```

Parameter Tuning — Initialization

	Zero	Random	Xavier
Pro			
Con			

Parameter Tuning — Initialization

	Zero	Random	Xavier
Pro	**Simple to implement**		
Con	**Symmetry problem:** Neurons in each layer will learn the same features during training Prevents the network from learning complex patterns.		

Parameter Tuning — Initialization

	Zero	Random	Xavier
Pro	**Simple to implement**	**Breaks symmetry** Can potentially lead to **faster convergence** if the scale of initialization is set appropriately.	
Con	**Symmetry problem:** Neurons in each layer will learn the same features during training Prevents the network from learning complex patterns.	Risk of **exploding or vanishing gradients**. Finding the right distribution and scale can be tricky and might **need experimentation.**	

Parameter Tuning — Initialization

	Zero	Random	Xavier
Pro	**Simple to implement**	**Breaks symmetry.** Can potentially lead to **faster convergence** if the scale of initialization is set appropriately.	**Controls the variance** of the outputs and gradients. By maintaining variance, it **helps avoid the vanishing and exploding gradients** problem.
Con	**Symmetry problem:** Neurons in each layer will learn the same features during training Prevents the network from learning complex patterns.	Risk of **exploding or vanishing gradients**. Finding the right distribution and scale can be tricky and might **need experimentation.**	Assumes linear activations. May not be suitable for very deep networks.

Parameter Tuning — Adaptive Learning Rate

Parameter Tuning — Adaptive Learning Rate

- It is important to choose the appropriate learning rate during training.

Parameter Tuning — Adaptive Learning Rate

- It is important to choose the appropriate learning rate during training.
- Letting the learning rate vary when training a model can reduce the training time and improve the numerical optimal solution.

Parameter Tuning — Adaptive Learning Rate

Method	Explanation	Update of w	Update of b
Momentum	• Dampens oscillations • Improvement to SGD • 2 parameters to tune	$w - \alpha v_{dw}$	$b - \alpha v_{db}$

Parameter Tuning — Adaptive Learning Rate

Method	Explanation	Update of w	Update of b
Momentum	• Dampens oscillations • Improvement to SGD • 2 parameters to tune	$w - \alpha v_{dw}$	$b - \alpha v_{db}$
RMSprop	• Root Mean Square propagation • Speeds up learning algorithm by controlling oscillations	$w - \alpha \frac{dw}{\sqrt{s_{dw}}}$	$b \longleftarrow b - \alpha \frac{db}{\sqrt{s_{db}}}$

Parameter Tuning — Adaptive Learning Rate

Method	Explanation	Update of w	Update of b
Momentum	• Dampens oscillations • Improvement to SGD • 2 parameters to tune	$w - \alpha v_{dw}$	$b - \alpha v_{db}$
RMSprop	• Root Mean Square propagation • Speeds up learning algorithm by controlling oscillations	$w - \alpha \frac{dw}{\sqrt{s_{dw}}}$	$b \longleftarrow b - \alpha \frac{db}{\sqrt{s_{db}}}$
Adam	• Adaptive Moment estimation • Most popular method • 4 parameters to tune	$w - \alpha \frac{v_{dw}}{\sqrt{s_{dw}} + \epsilon}$	$b \longleftarrow b - \alpha \frac{v_{db}}{\sqrt{s_{db}} + \epsilon}$

Parameter Tuning — Adaptive Learning Rate

Method	Explanation	Update of w	Update of b
Momentum	• Dampens oscillations • Improvement to SGD • 2 parameters to tune	$w - \alpha v_{dw}$	$b - \alpha v_{db}$
RMSprop	• Root Mean Square propagation • Speeds up learning algorithm by controlling oscillations	$w - \alpha \frac{dw}{\sqrt{s_{dw}}}$	$b \longleftarrow b - \alpha \frac{db}{\sqrt{s_{db}}}$
Adam	• Adaptive Moment estimation • Most popular method • 4 parameters to tune	$w - \alpha \frac{v_{dw}}{\sqrt{s_{dw}} + \epsilon}$	$b \longleftarrow b - \alpha \frac{v_{db}}{\sqrt{s_{db}} + \epsilon}$

Regularization — Weight Regularization

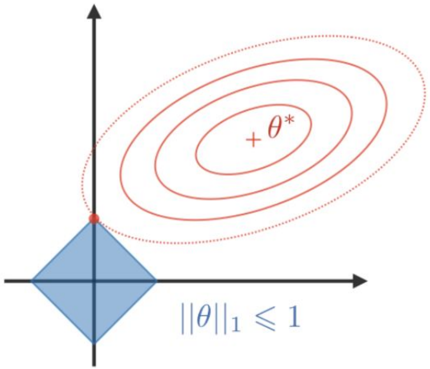
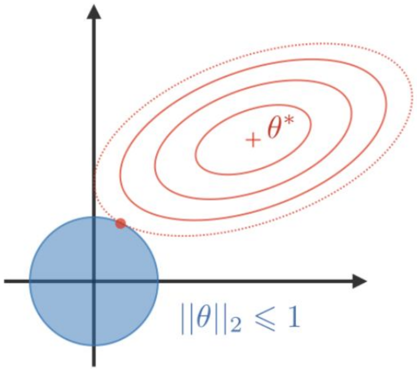
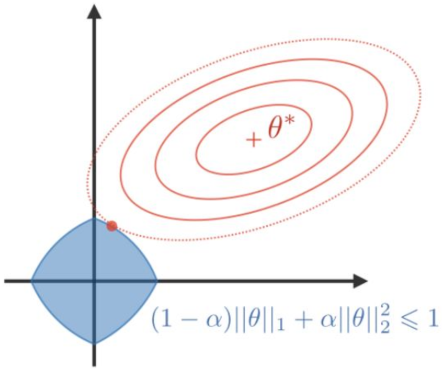
Regularization — Weight Regularization

- Overfitting is a common problem during training.

Regularization — Weight Regularization

- Overfitting is a common problem during training.
- In order to make sure that the weights are not too large and that the model is not overfitting the training set, regularization techniques are usually performed on the model weights.

Regularization — Weight Regularization

LASSO	Ridge	Elastic Net
• Shrinks coefficients to 0 • Good for variable selection	Makes coefficients smaller	Tradeoff between variable selection and small coefficients
		
$... + \lambda\|\|\theta\|\|_1$ $\lambda \in \mathbb{R}$	$... + \lambda\|\|\theta\|\|_2^2$ $\lambda \in \mathbb{R}$	$... + \lambda\Big[(1-\alpha)\|\|\theta\|\|_1 + \alpha\|\|\theta\|\|_2^2\Big]$ $\lambda \in \mathbb{R}, \alpha \in [0,1]$

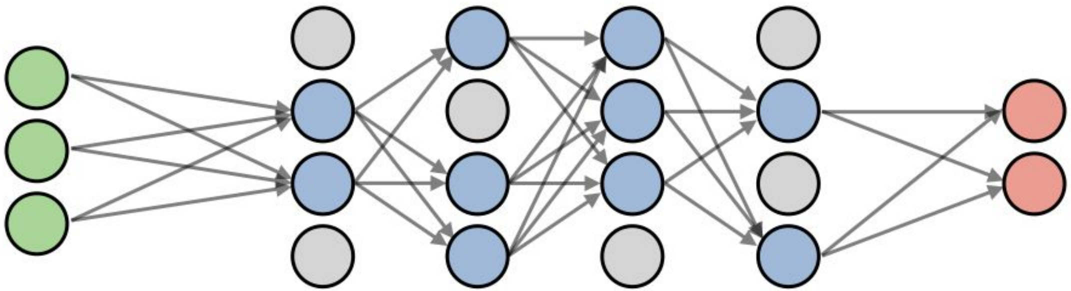
Regularization — Dropout

Regularization — Dropout

- Dropout is a technique used in neural networks to prevent overfitting the training data by dropping out neurons with probability $1 - p > 0$, where p is the probability to keep the neuron. (Note: be careful about the definition of p.)

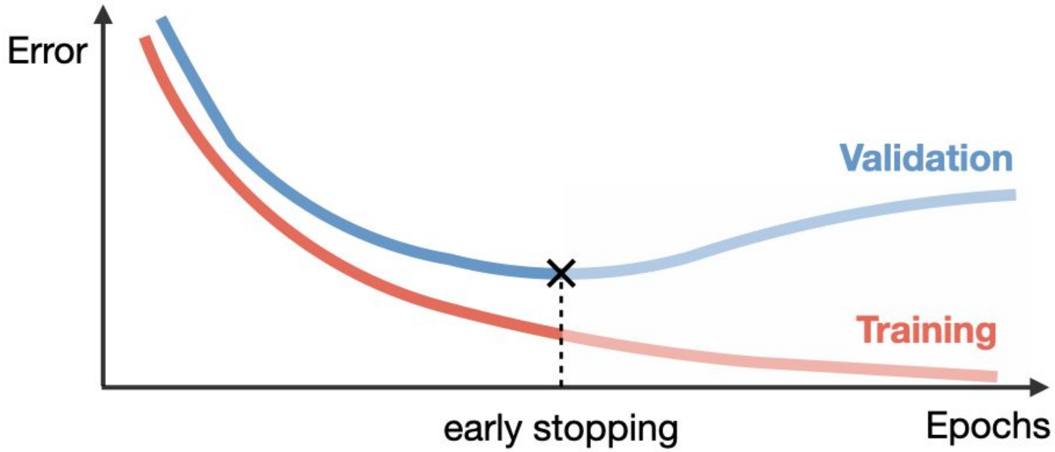
Regularization — Dropout

- Dropout is a technique used in neural networks to prevent overfitting the training data by dropping out neurons with probability $1 - p > 0$, where p is the probability to keep the neuron. (Note: be careful about the definition of p.)
- It forces the model to avoid relying too much on particular sets of features.



Regularization — Early Stop

- This regularization technique stops the training process as soon as the validation loss reaches a plateau or starts to increase.



Good Practices — Overfitting Small Batch

- When debugging a model, it is often useful to make quick tests to see if there is any major issue with the architecture of the model itself.

Good Practices — Overfitting Small Batch

- When debugging a model, it is often useful to make quick tests to see if there is any major issue with the architecture of the model itself.
- In particular, in order to make sure that the model can be properly trained, a mini-batch is passed inside the network to see if it can overfit on it.

Good Practices — Overfitting Small Batch

- When debugging a model, it is often useful to make quick tests to see if there is any major issue with the architecture of the model itself.
- In particular, in order to make sure that the model can be properly trained, a mini-batch is passed inside the network to see if it can overfit on it.
- If it cannot, it means that the model is either too complex or not complex enough (most likely) to even overfit on a small batch, let alone a normal-sized training set.

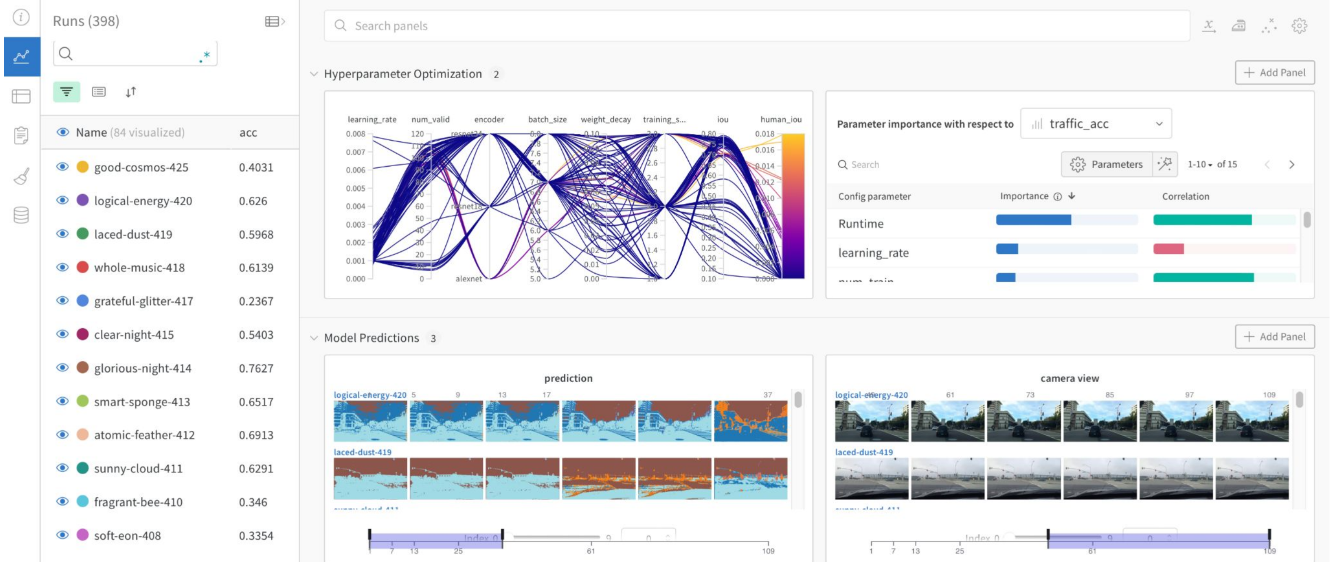
Good Practices — Weights & Biases

Good Practices — Weights & Biases

- The main use of the wandb library is to **track and visualize the different machine learning experiments**, the training process, the hyperparameters and the models.

Good Practices — [Weights & Biases]

- The main use of the wandb library is to **track and visualize the different machine learning experiments**, the training process, the hyperparameters and the models.



Good Practices — Weights & Biases

```
for epoch in range(10):
        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
           inputs, labels = data[0].to(device), data[1].to(device)
           optimizer.zero_grad()
           outputs = net(inputs)
           loss = criterion(outputs, labels)
           loss.backward()
           optimizer.step()
           running_loss += loss.item()

           if i % 2000 == 1999:    # print every 2000 mini-batches
              print('[%d, %5d] loss: %.3f' %
                   (epoch + 1, i + 1, running_loss / 2000))

              running_loss = 0.0
```

Good Practices — Weights & Biases

```
for epoch in range(10):
        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
           inputs, labels = data[0].to(device), data[1].to(device)
           optimizer.zero_grad()
           outputs = net(inputs)
           loss = criterion(outputs, labels)
           loss.backward()
           optimizer.step()
           running_loss += loss.item()

           if i % 2000 == 1999:    # print every 2000 mini-batches
              print('[%d, %5d] loss: %.3f' %
                   (epoch + 1, i + 1, running_loss / 2000))
              wandb.log({'epoch': epoch+1, 'loss':
     running_loss/2000})
              running_loss = 0.0
```

Good Practices — [Weights & Biases]

```
for epoch in range(10):
        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
           inputs, labels = data[0].to(device), data[1].to(device)
           optimizer.zero_grad()
           outputs = net(inputs)
           loss = criterion(outputs, labels)
           loss.backward()
           optimizer.step()
           running_loss += loss.item()

           if i % 2000 == 1999:    # print every 2000 mini-batches
              print('[%d, %5d] loss: %.3f' %
                   (epoch + 1, i + 1, running_loss / 2000))
              wandb.log({'epoch': epoch+1, 'loss':
     running_loss/2000})
              running_loss = 0.0
```