CS 498: Machine Learning System Spring 2025

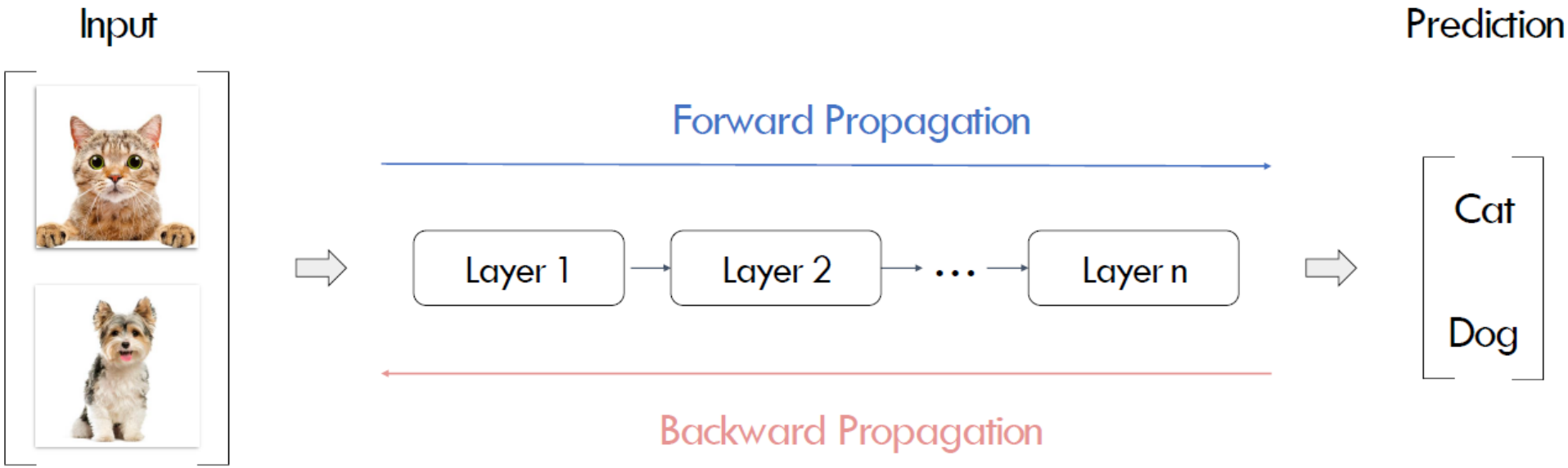
Minjia Zhang

The Grainger College of Engineering

Today

- **Understand the workloads: Deep Learning**
  - **Models**
  - **Optimization**
  - **Frameworks**
- **Hardware**

Background: DL Computation



$$\theta^{(t+1)} = f\left(\theta^{(t)}, \nabla_L\left(\theta^{(t)}, D^{(t)}\right)\right)$$

parameter — weight update (sgd, adam, etc.) — model (CNN, GPT, etc.) — data

Three Important Components

Data

- images
- Text
- Audio
- Table
- etc.

Model

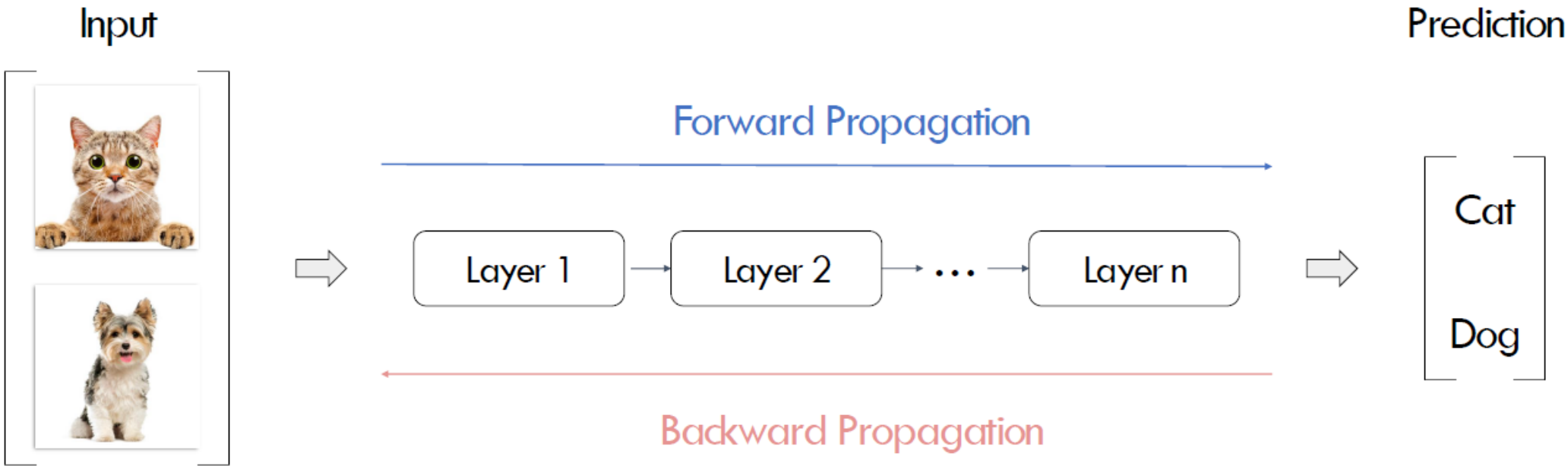
- CNNs
- RNNs
- GNNs
- Transformers
- MoEs

Compute

- cpus
- gpus/tpus/lpus
- M3/FPGA/etc.

How to express these computation?

- Idea: Composable Layers



Understand Our Workload

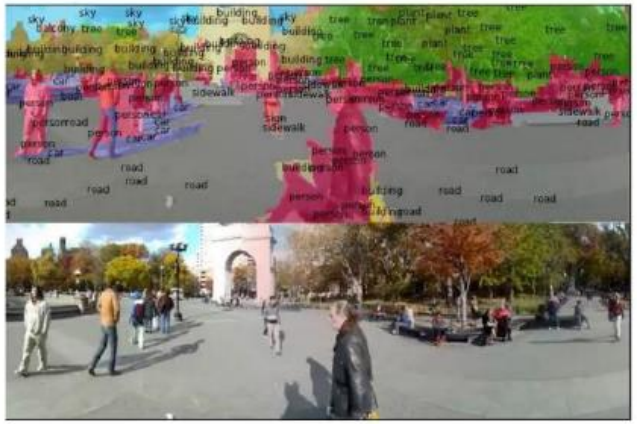
- There are many great models developed in the history
- In this class, we review the most important 5 classes
  - Convolutional Neural Networks
  - Recurrent neural networks
  - Transformers
  - Graph neural networks
  - Mixture-of-Experts

CNNs: Applications

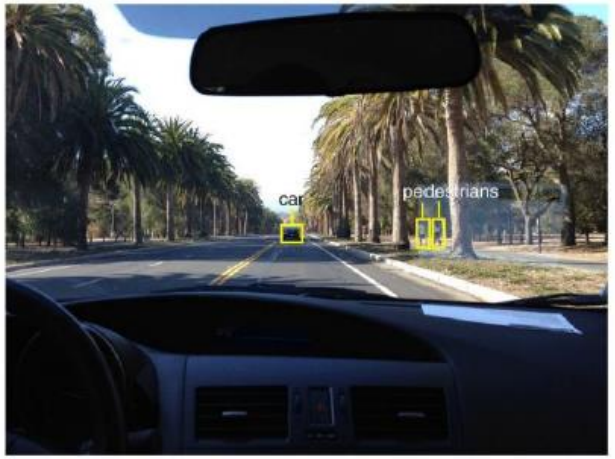
Classification

Retrieval

Detection



Segmentation

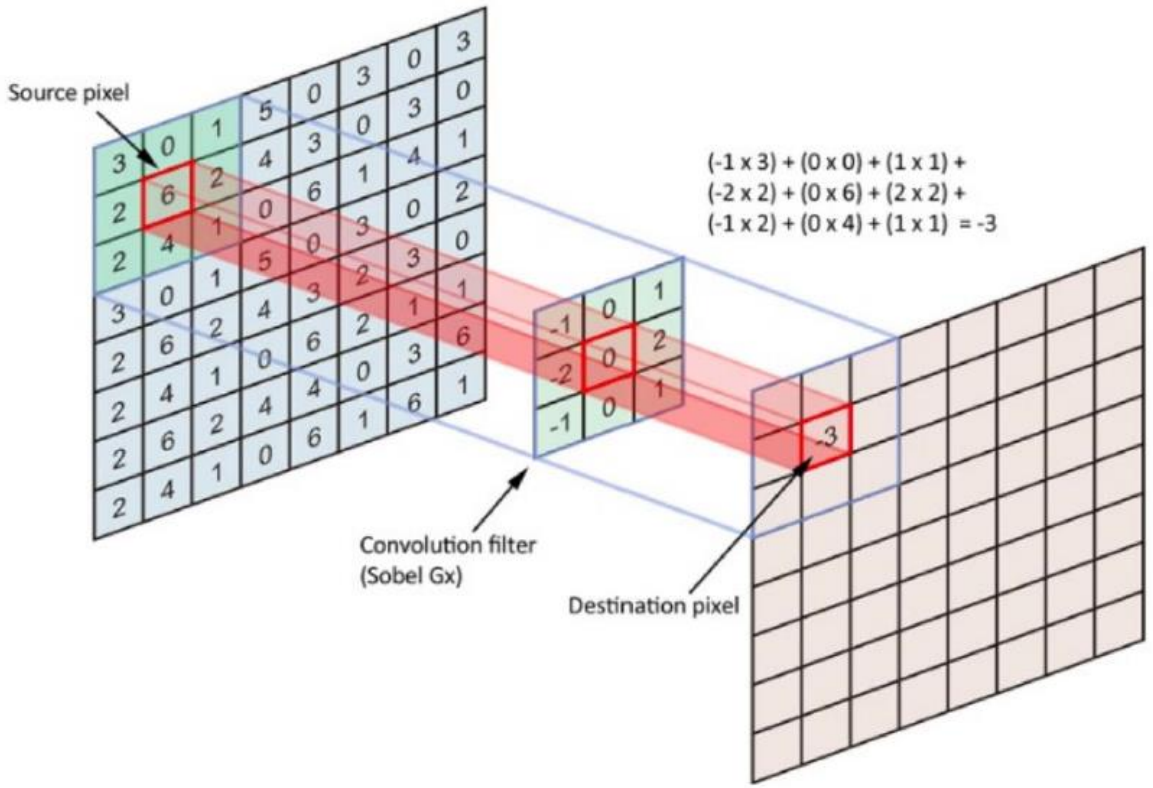


Self-Driving

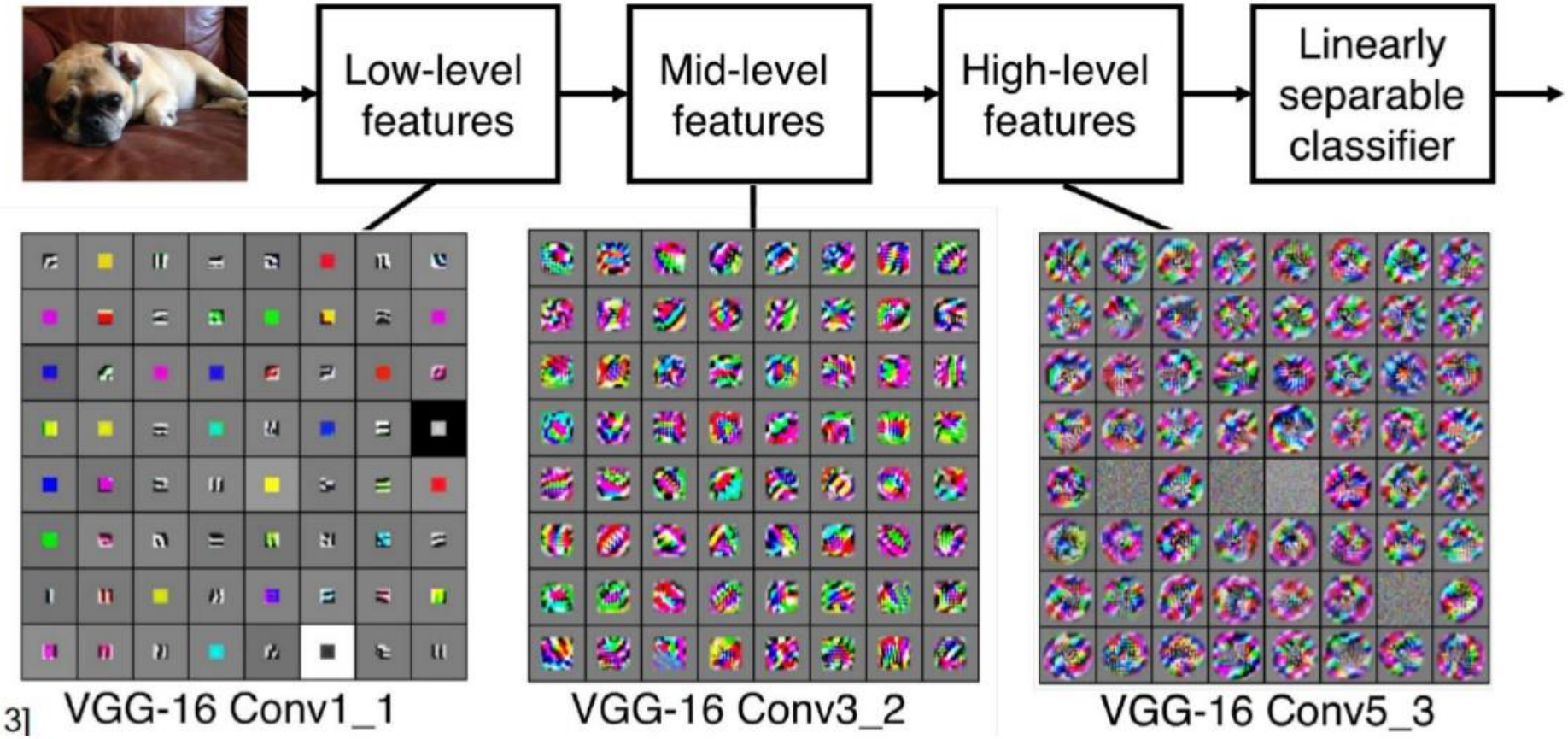
Synthesis

CNNs: Key Components

- Convolve the filter with the image: slide over the image spatially and compute dot products



Stacking Conv Layers



Zeiler and Fergus 2013]

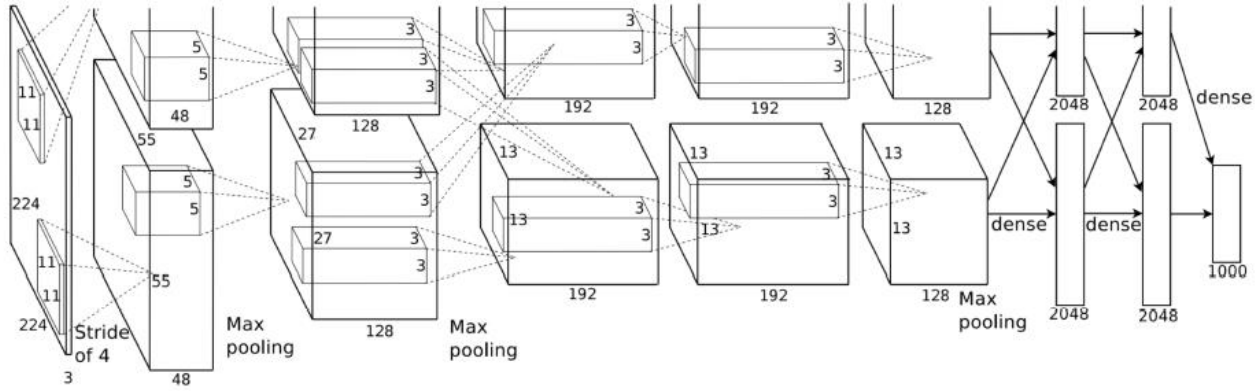
Edges, corners, textures

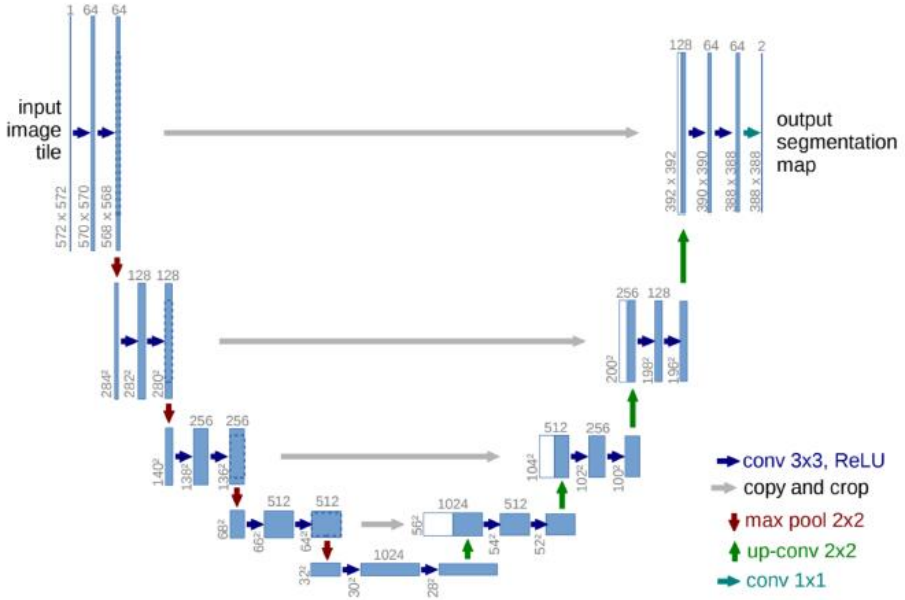
More complex patterns: shapes and parts of objects

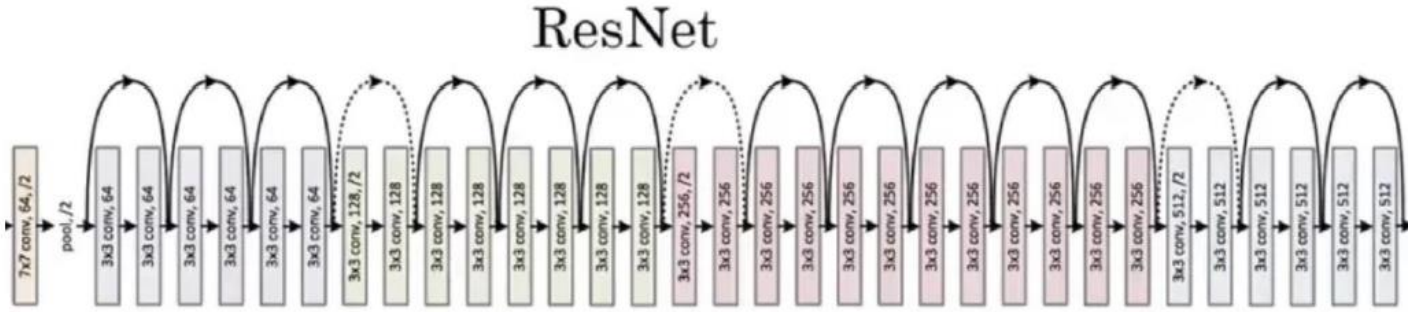
Abstract features: objects, faces,…

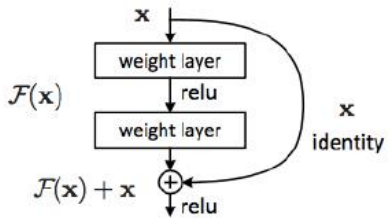
CNN: Top3 Models

- AlexNet by Alex/Iliya/Hinton
- ResNet by Kaiming etc.
- U-Net by Olaf etc.





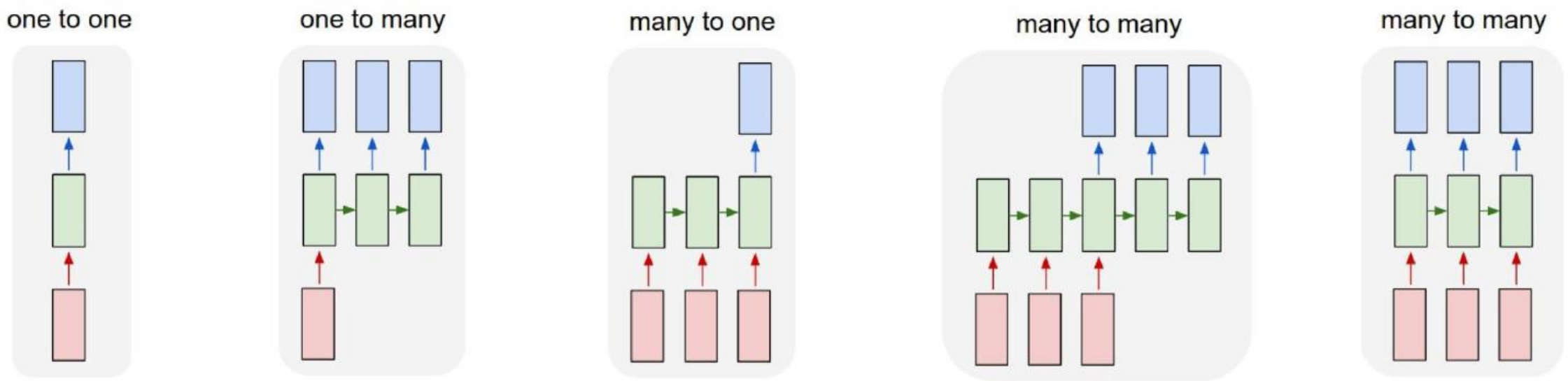




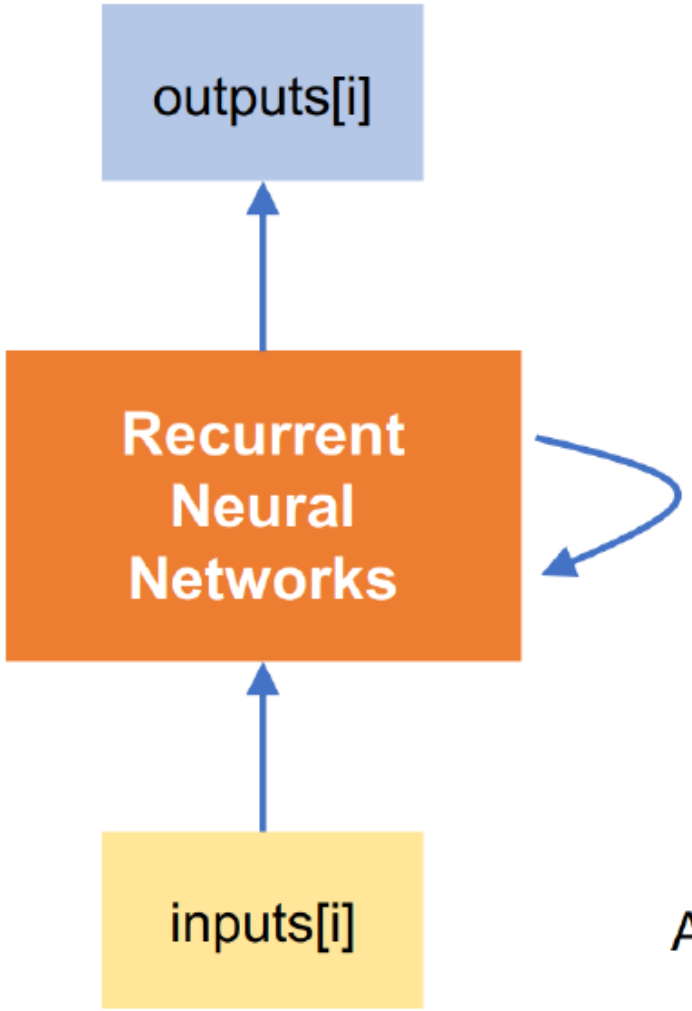
CNN more important components

- Conv
  - Conv1d, Conv2d, conv3d, etc.
- Matmul (linear) :
  - C = A * B
  - Softmax
- Elementwise operations:
  - ReLU, add, sub
- Other ops
  - Pooling, normalization, etc.

Sequence Modeling



Recurrent Neural Networks

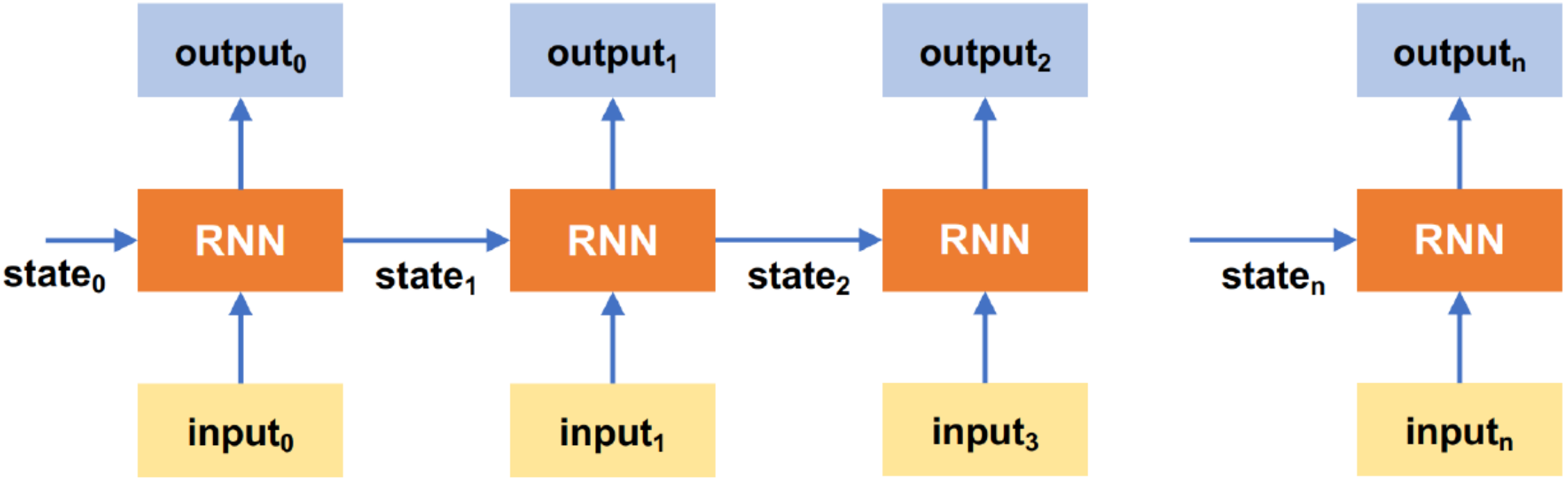


Arbitrary number of outputs

Key idea: RNNs have an internal state that is updated as a sequence is processed

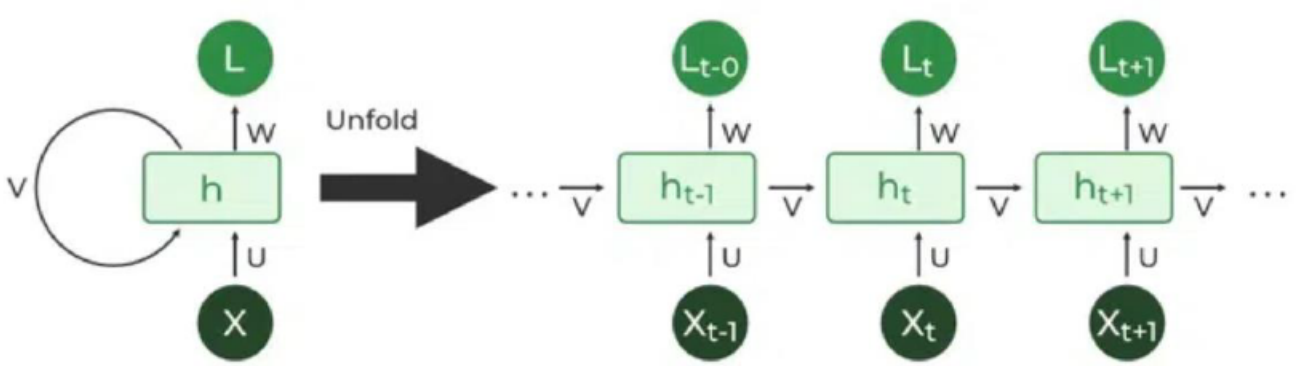
Arbitrary number of inputs

Recurrent Neural Networks: Unrolling the Computation



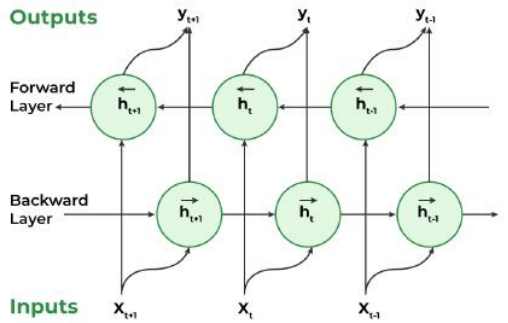
Most Important Components in RNNs

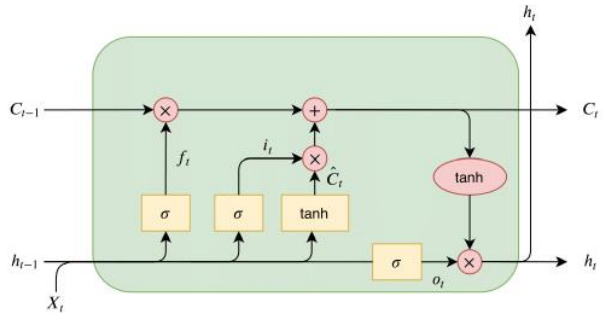
- One can make any basic neural network recurrent
- Matmul
- Elementwise nonlinear
  - ReLU, Tanh, sigmoid, etc.

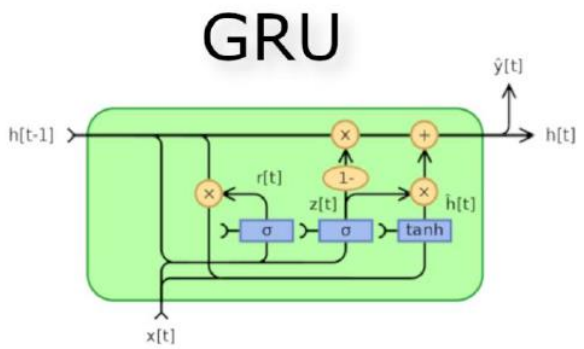


RNN: Top3 Models

- Bidirectional RNNs
- LSTM
- GRU

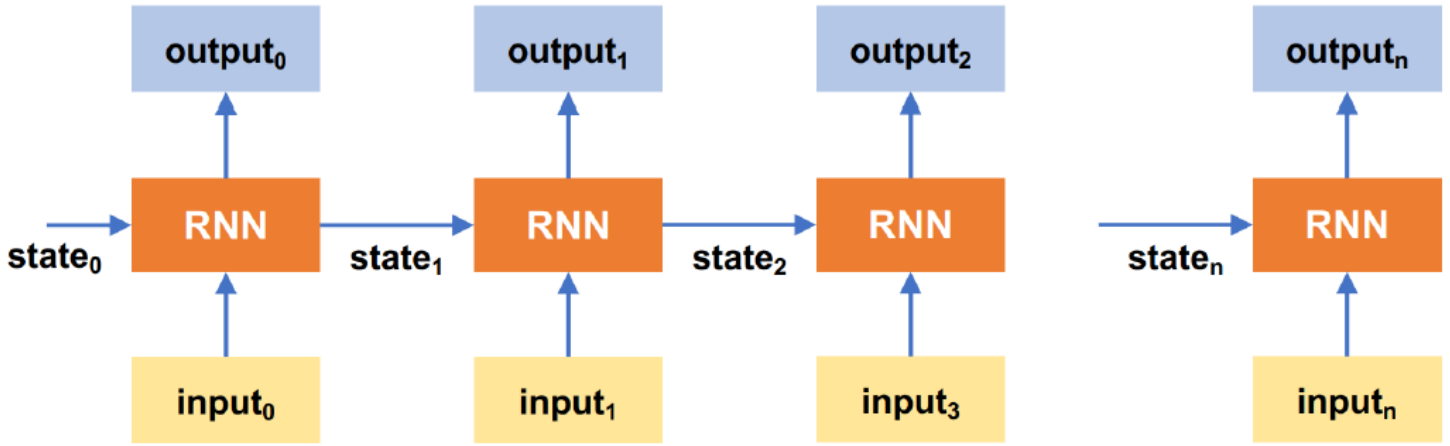






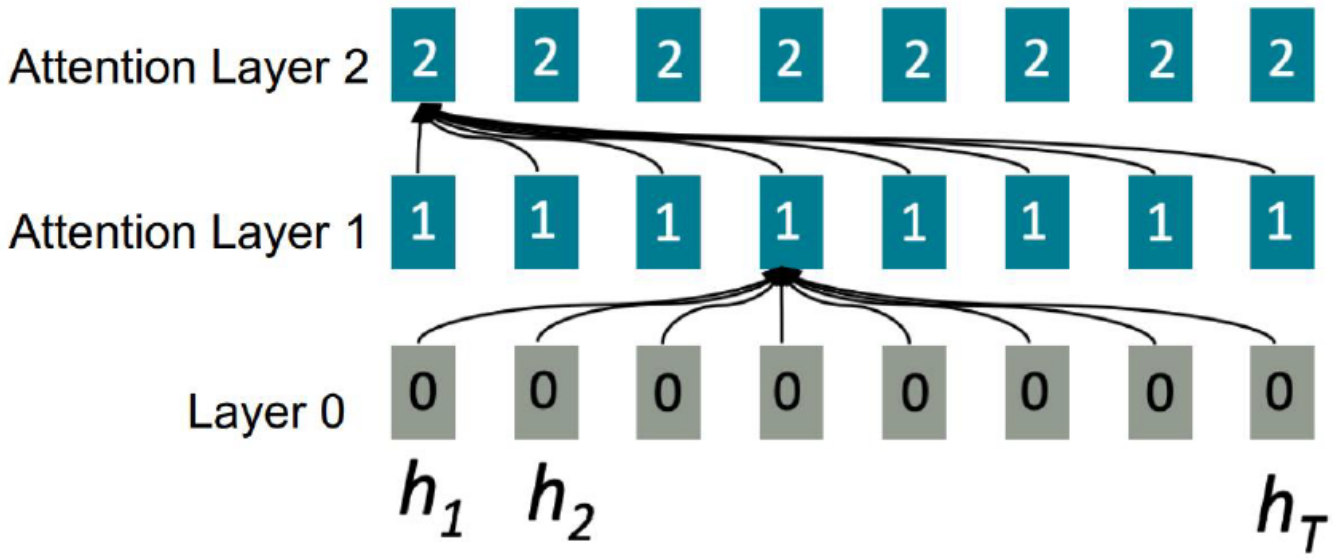
Two Key Problems of RNNs

- Problem 1: **lack of parallelizability**.
  - Both forward and backward passes have O(sequence length) unparallelizable operators
  - A state cannot be computed before all previous states have been computed Inhibits training on very long sequence
- Problem 2: forgetting.



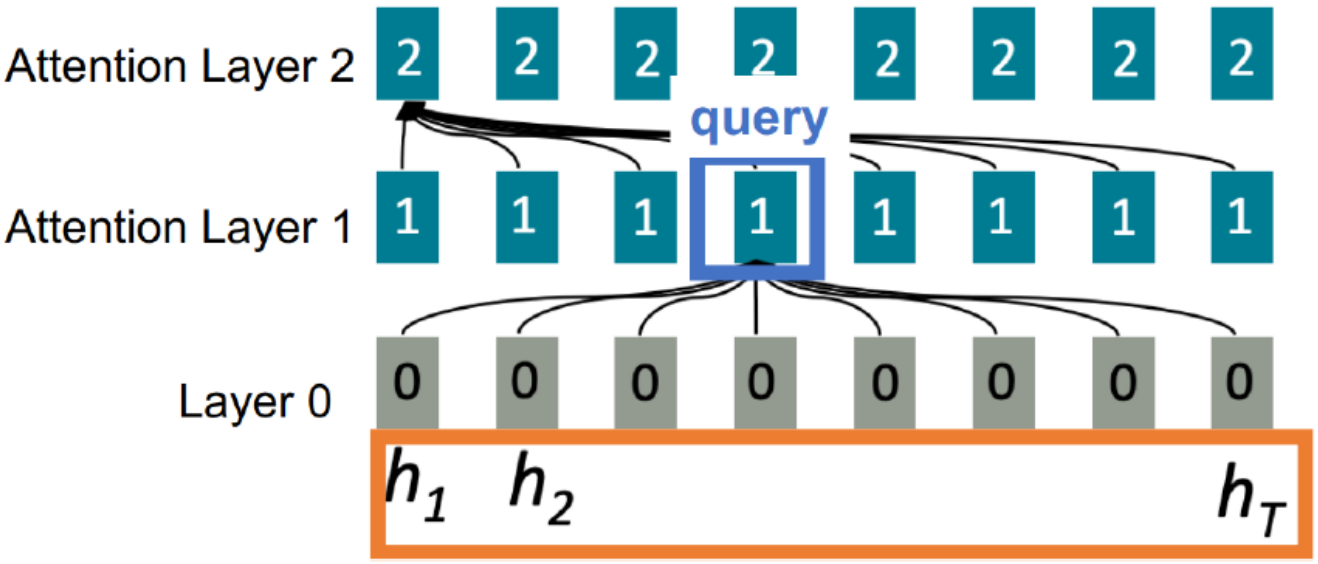
Attention: Enable parallelism

- Idea: treat each position's representation as a query to access and incorporate information from a set of values



Attention

- Massively parallelizable: number of unparallelizable operations does not increase sequence length



We will learn attention and transformers in depth later:

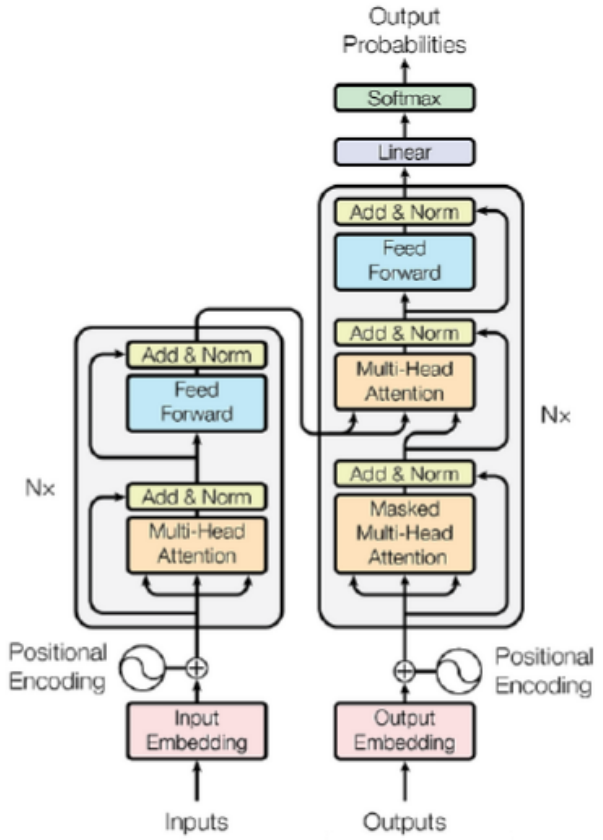
- Self-attention
- Masked attention
- Multi-head attention

Transformers

- Transformer = attention + a few MLPs

BERT

Encoder



GPT

Decoder

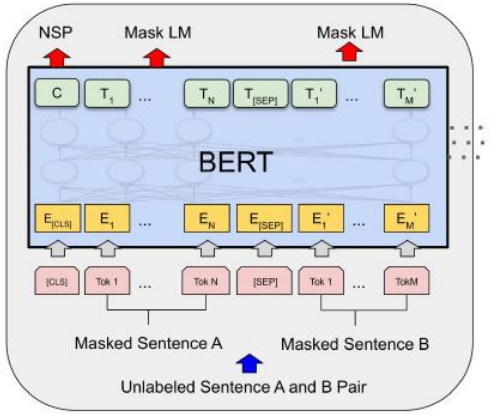
[Recommended reading: The Illustrated Transformer – Jay Alammar – Visualizing machine learning one concept at a time.]

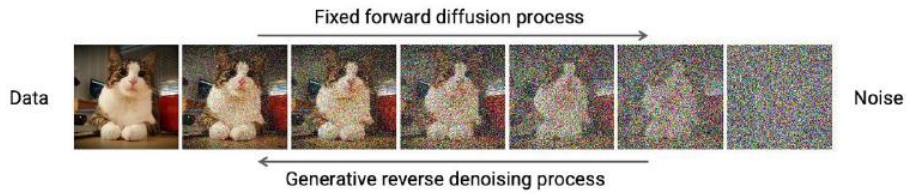
Most Important Components in attentions

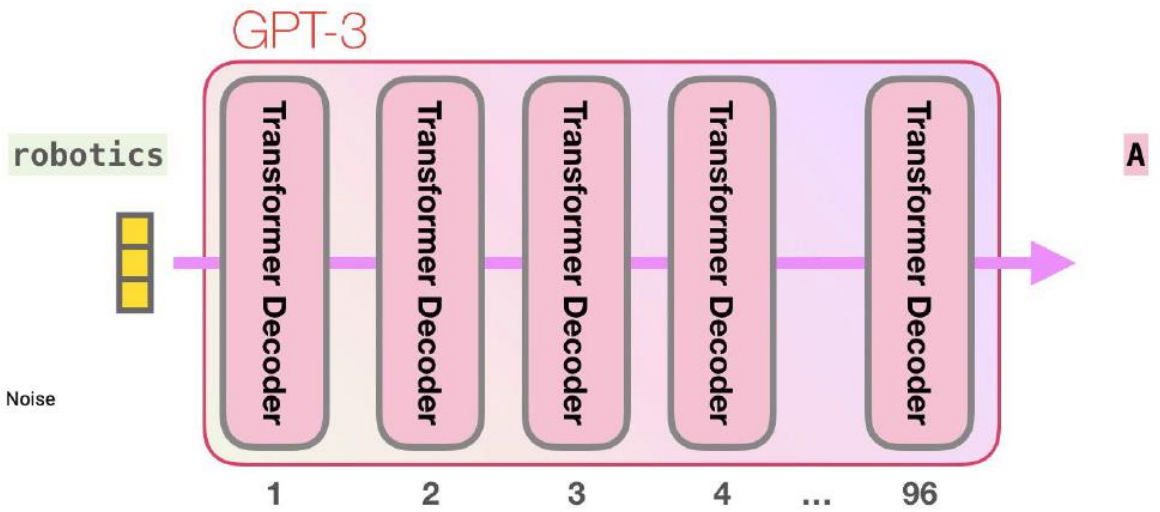
- Attention, which is composed by a set of
  - Matmul
  - Softmax
  - Normalization

Top3 Transformers

- Bert
- GPT/LLMs
- DiT: diffusion



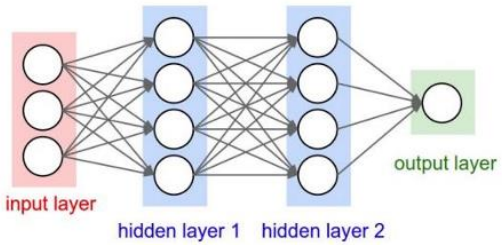




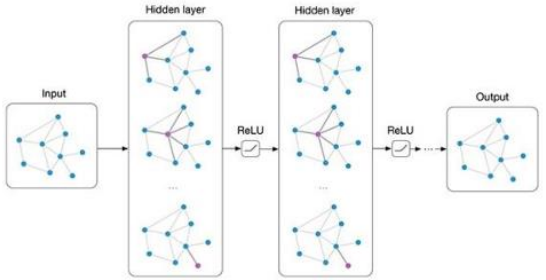
Graph Neural Networks

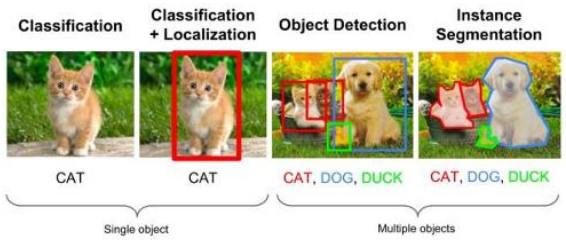
- Goal: model graph data

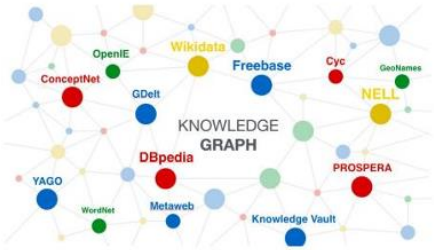
Neural Networks



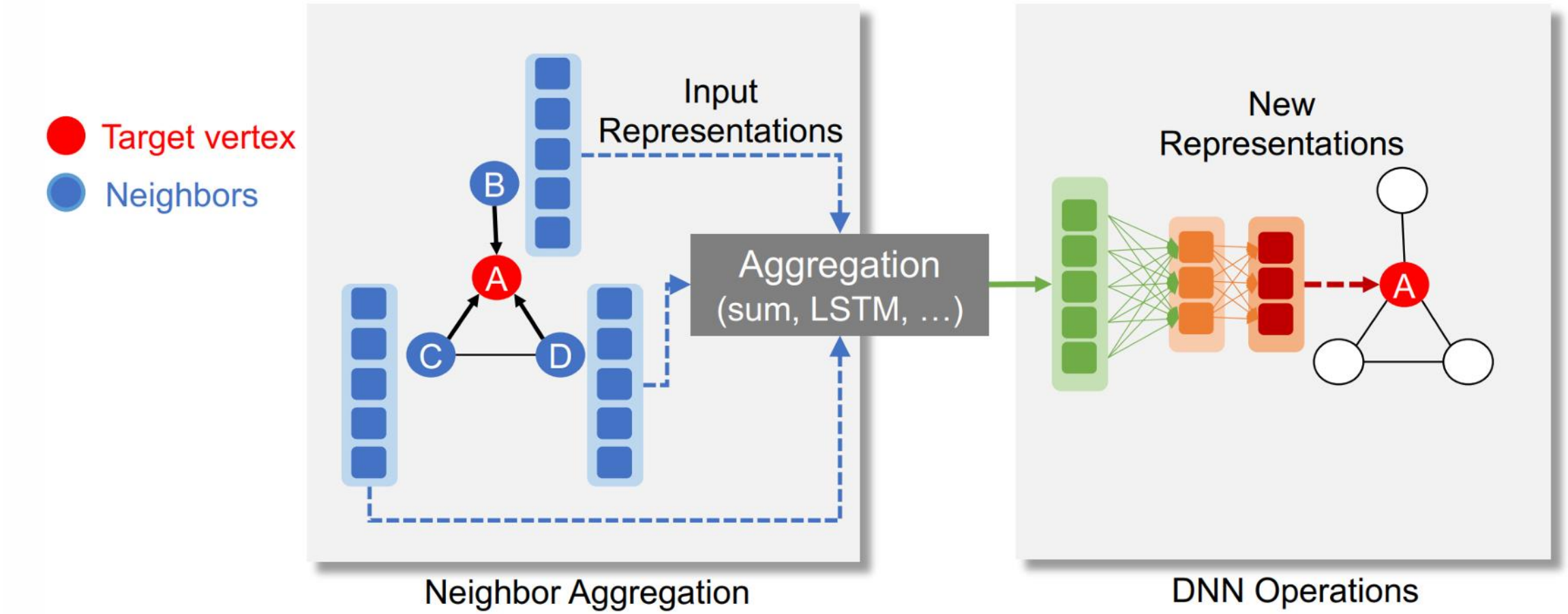
Graph Neural Networks



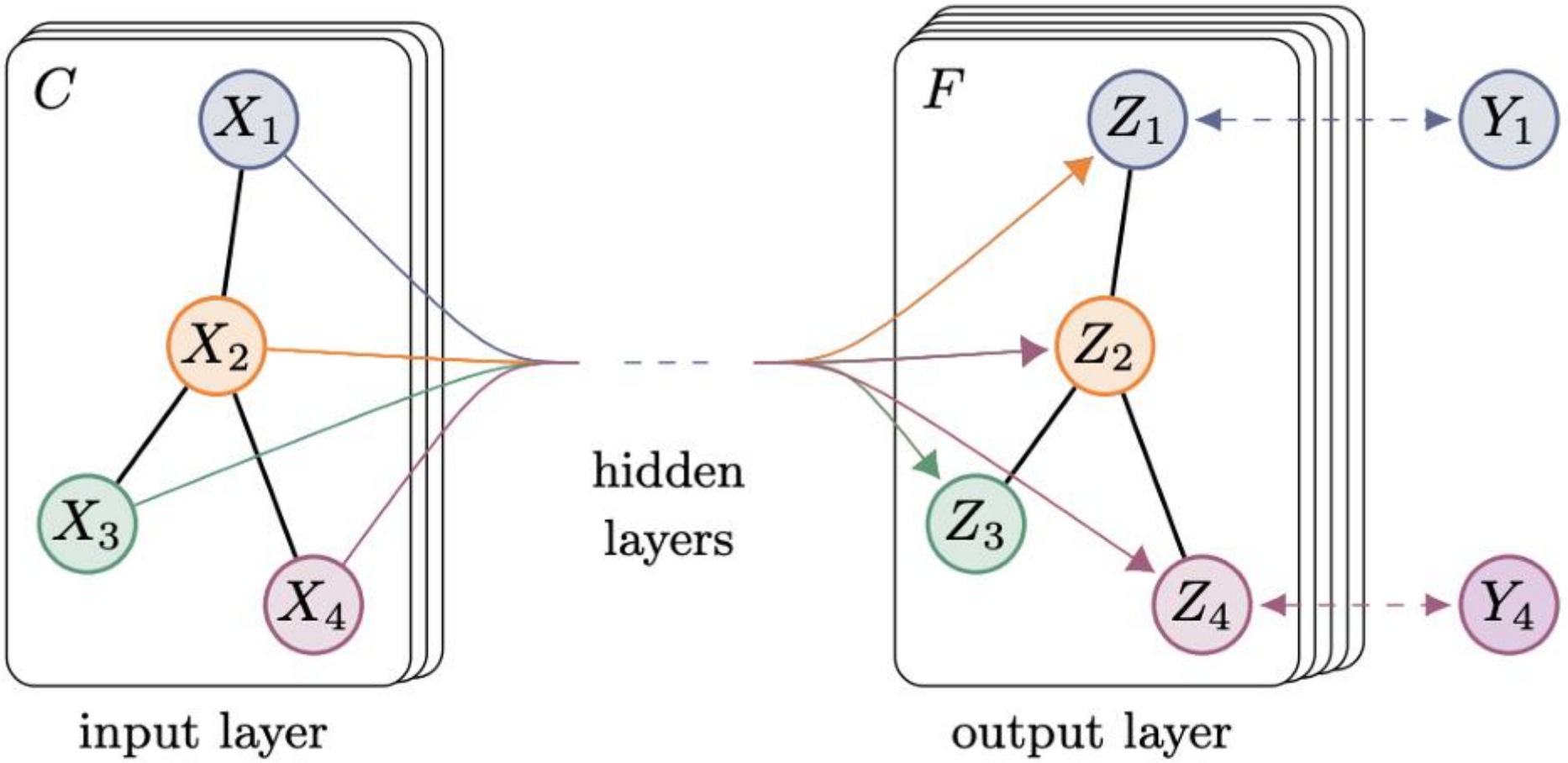




GNN Architecture

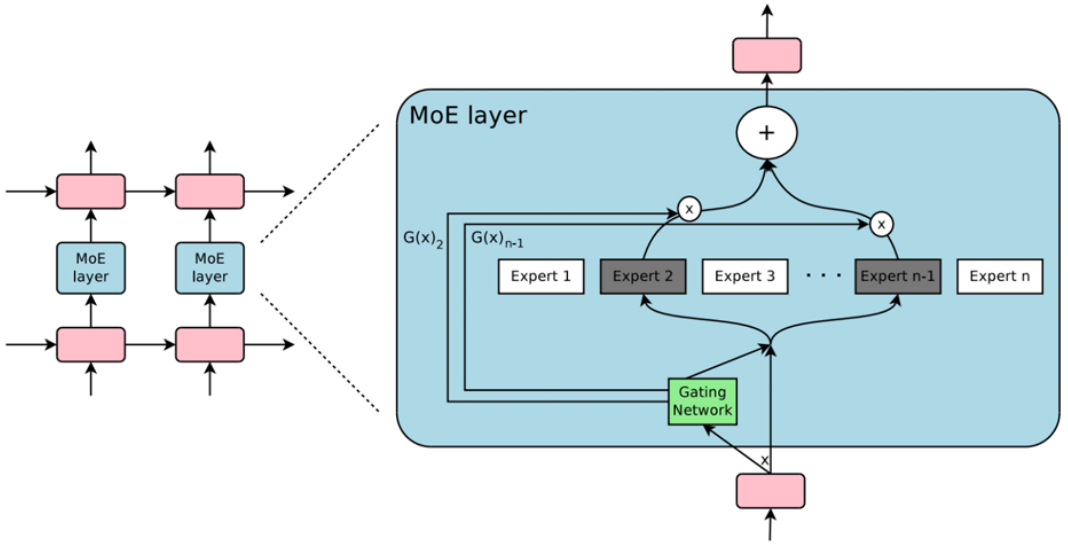


Top-1 GNNs: GCN Graph convolutional Networks



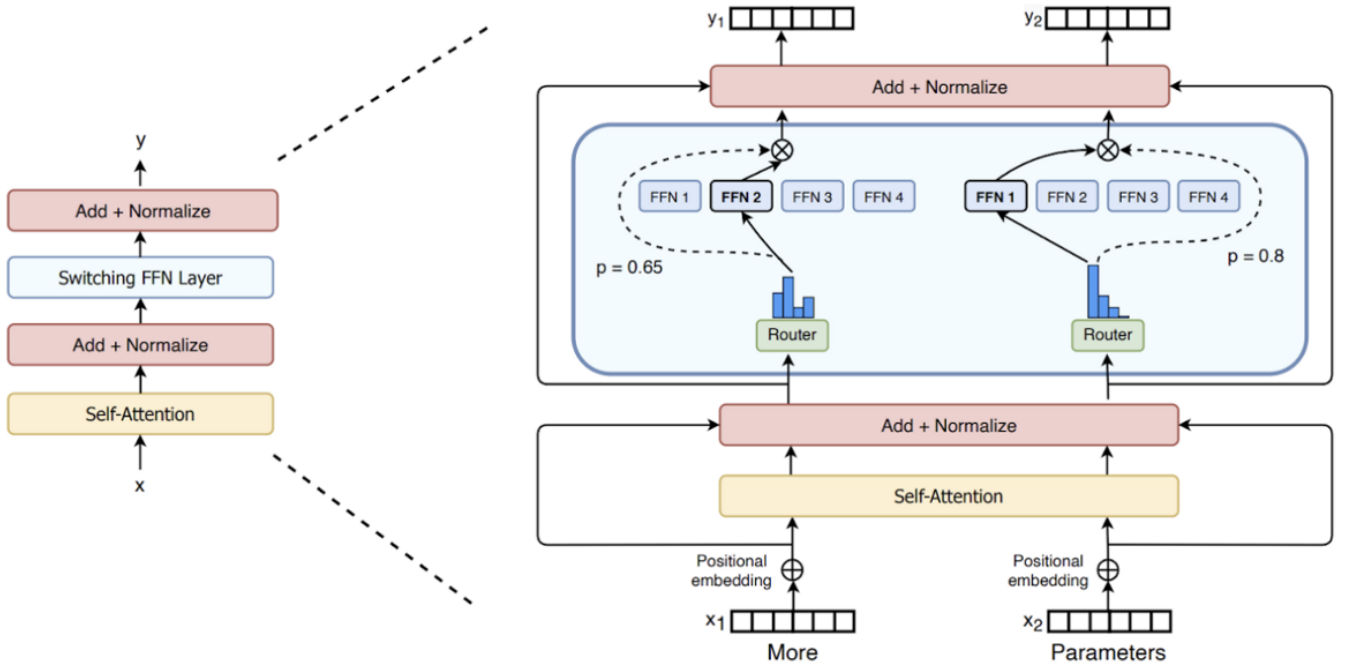
MoE: mixture of experts

- Ideas: More persons voting might be better than one person dictating
- Method: make each expert focus on predicting the right answer for a subset of cases



Novel Component in MoE

- Novel Components in MoE:
  - Router
- After-class Q:
  - Why router makes it hard



Background: DL Optimization

DL Optimization: Non-Convex Optimization

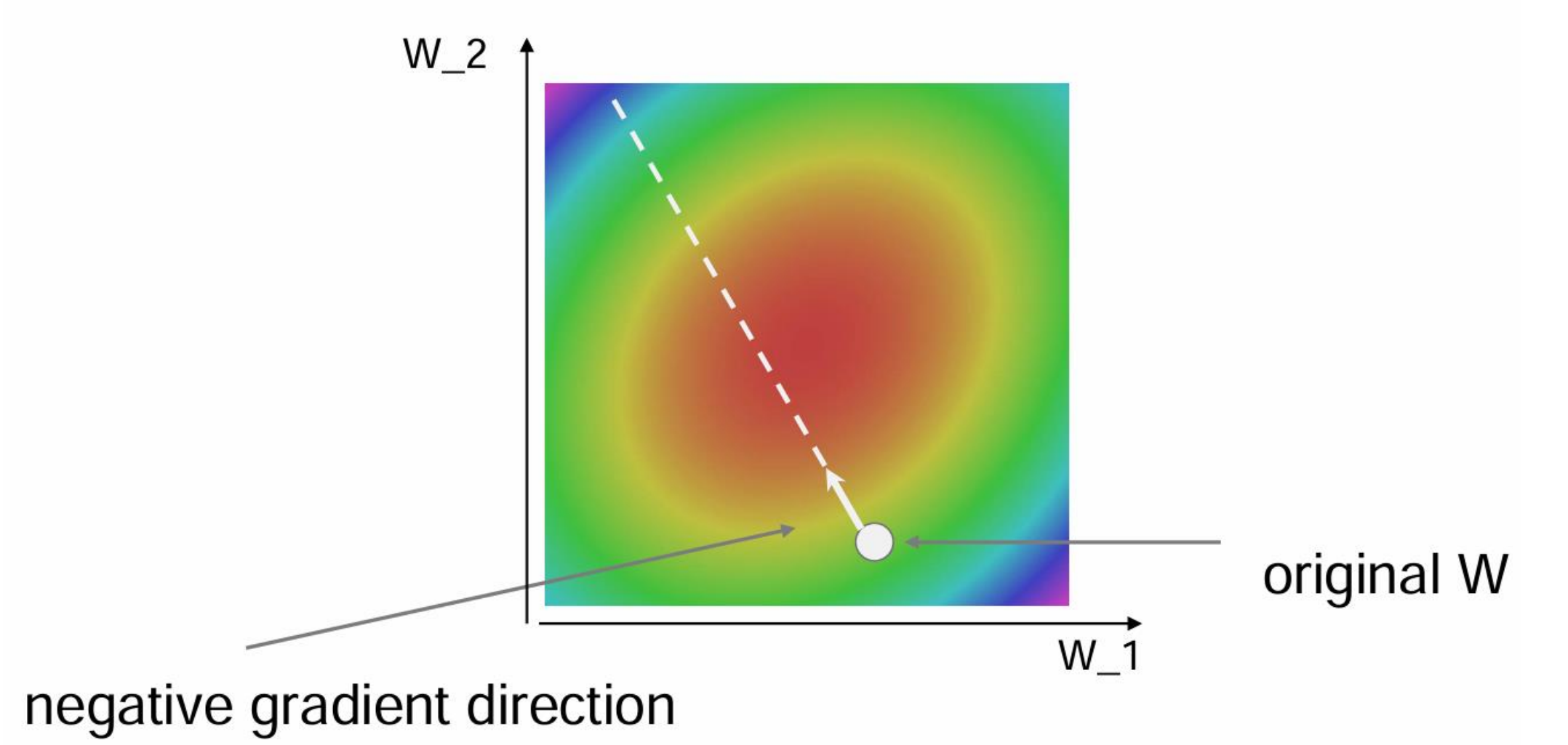
$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

Background: Gradient Descent

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Background: DL Optimization



Stochastic Gradient Descent (SGD)

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

Full sum expensive when N is large!

Approximate sum using a **minibatch** of examples
32 / 64 / 128 common

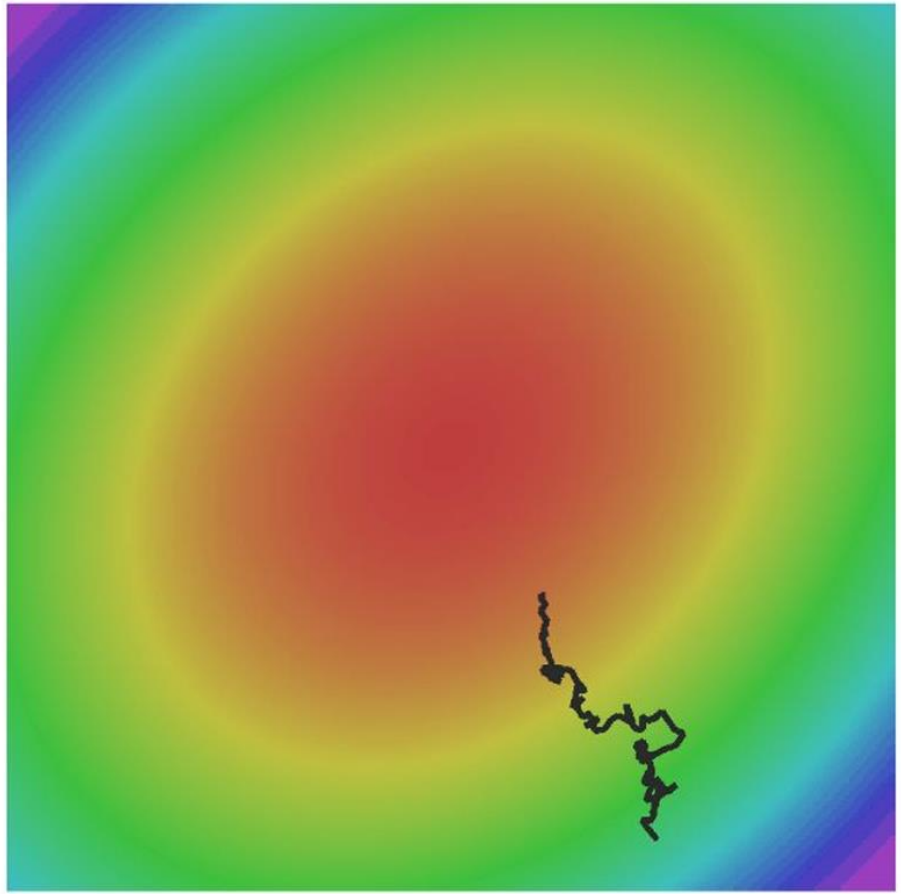
```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Optimization: Problems with SGD

Our gradients come from minibatches so they can be noisy!

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$



What if loss changes quickly in one direction and slowly in another?

SGD + Momentum

Continue moving in the general direction as the previous iterations

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x += learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

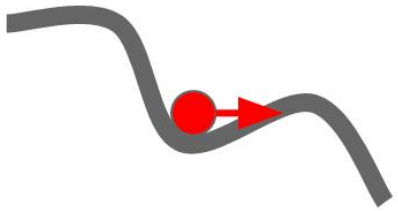
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

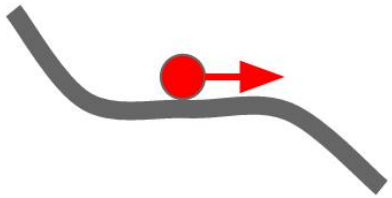
- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

SGD + Momentum

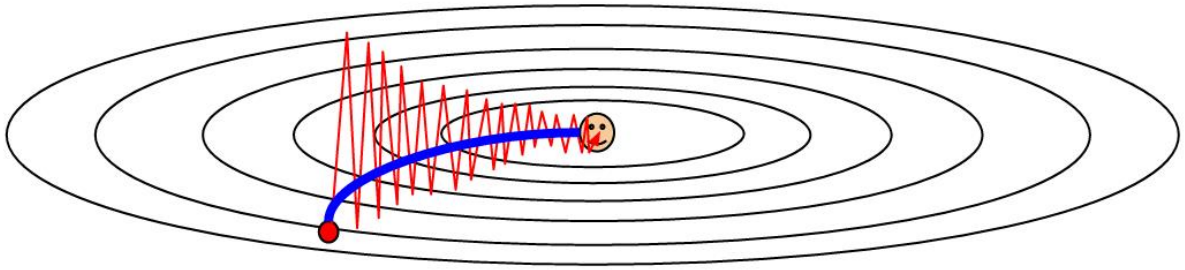
Local Minima



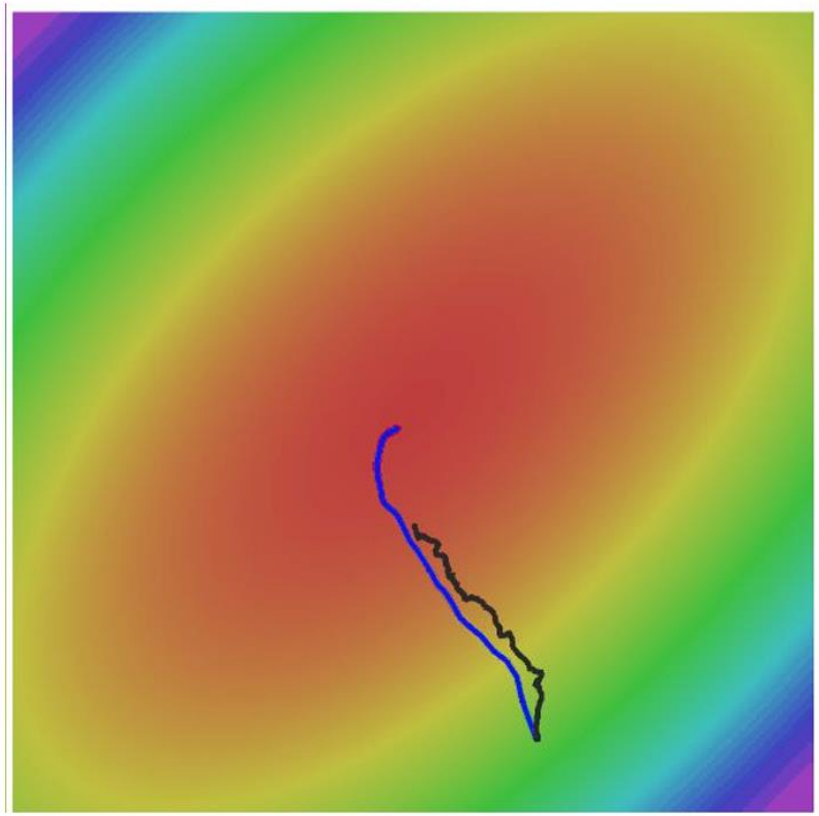
Saddle points



Poor Conditioning



Gradient Noise



More Complex Optimizers: RMSProp

SGD + Momentum

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

Adds element-wise scaling of the gradient based on the historical sum of squares in each dimension (with decay)

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Tieleman and Hinton, 2012

More Complex Optimizers: RMSProp

SGD + Momentum

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

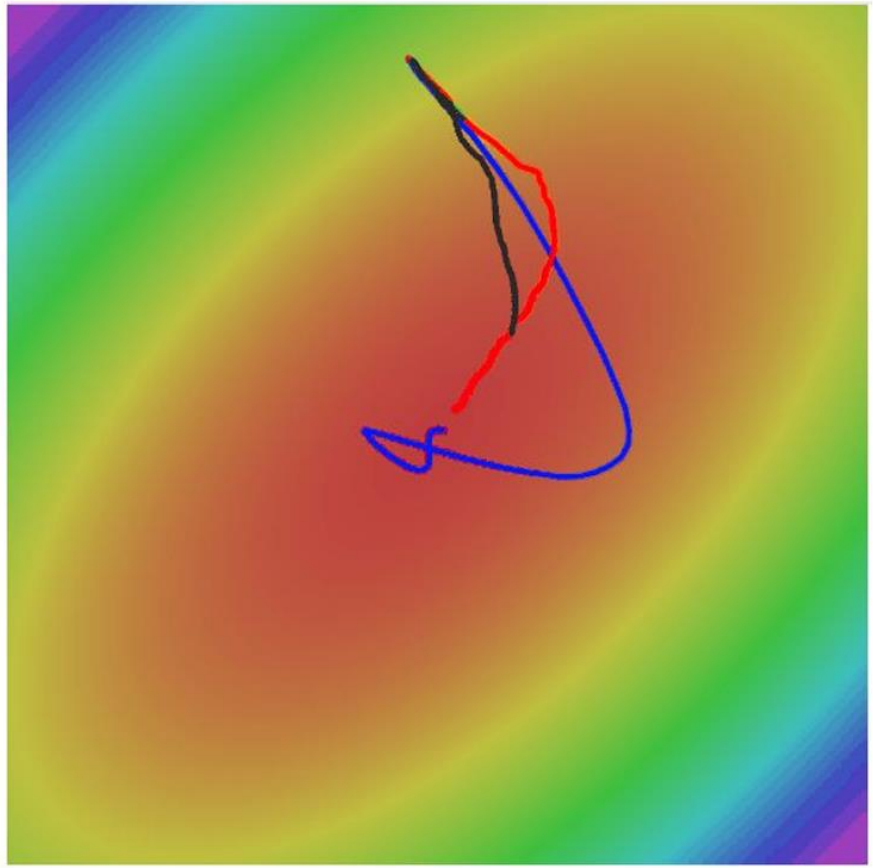
"Per-parameter learning rates" or "adaptive learning rates"

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Tieleman and Hinton, 2012

RMSProp



SGD

SGD+Momentum

RMSProp

Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Momentum

RMSProp

Sort of like RMSProp with momentum

Q: What happens at first timestep?

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

AdamW

Algorithm 2 Adam with L_2 regularization and Adam with decoupled weight decay (AdamW)

given $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
initialize time step $t \leftarrow 0$, parameter vector $\boldsymbol{\theta}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
repeat
  $t \leftarrow t + 1$
  $\nabla f_t(\boldsymbol{\theta}_{t-1}) \leftarrow \text{SelectBatch}(\boldsymbol{\theta}_{t-1})$ ▷ select batch and return the corresponding gradient
  $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{\theta}_{t-1}) + \lambda \boldsymbol{\theta}_{t-1}$
  $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t$ ▷ here and below all operations are element-wise
  $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t^2$
  $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1 - \beta_1^t)$ ▷ β_1 is taken to the power of t
  $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1 - \beta_2^t)$ ▷ β_2 is taken to the power of t
  $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ can be fixed, decay, or also be used for warm restarts
  $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \left(\alpha \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) + \lambda \boldsymbol{\theta}_{t-1} \right)$
until *stopping criterion is met*
return optimized parameters $\boldsymbol{\theta}_t$

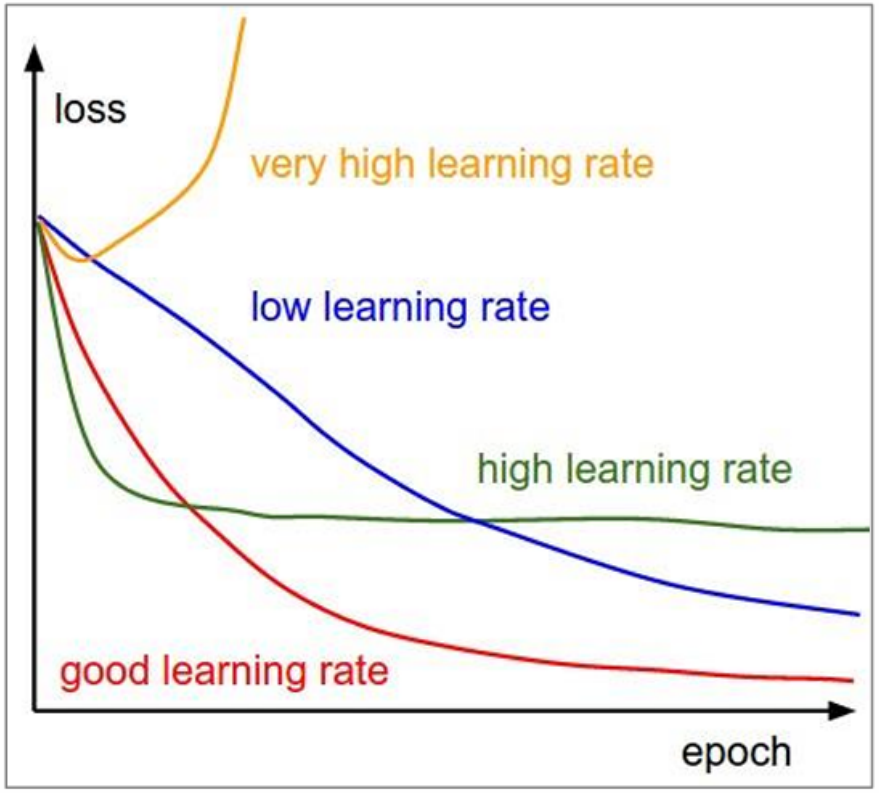
Learning Rate Schedules

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Learning rate

Learning Rate Schedules

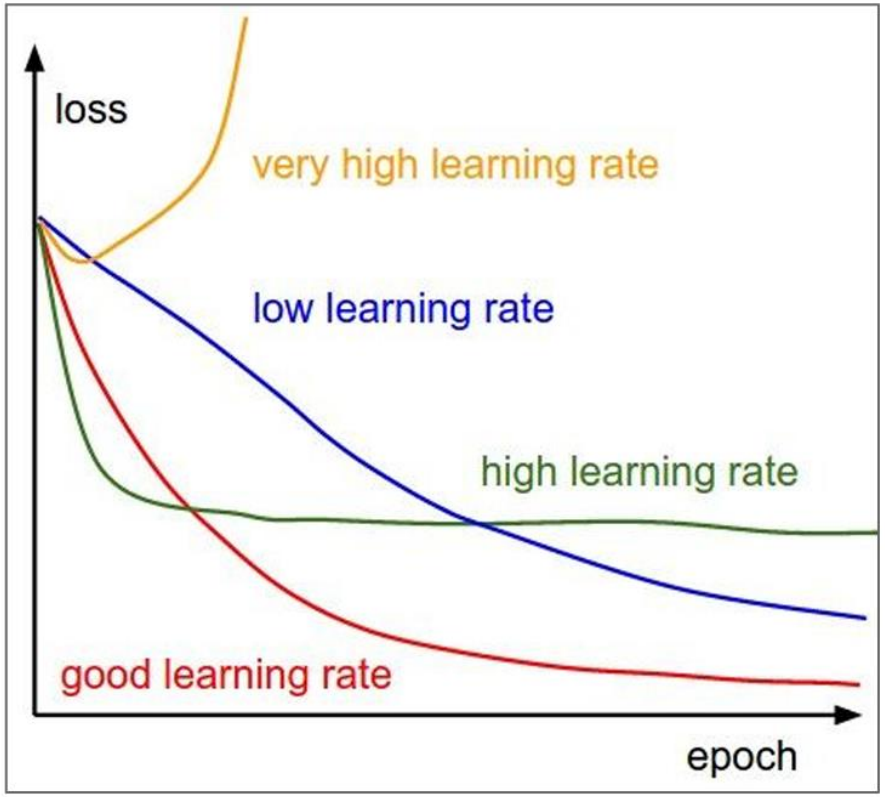
SGD, SGD+Momentum, RMSProp, Adam, AdamW all have **learning rate** as a hyperparameter.



Q: Which one of these learning rates is best to use?

Learning Rate Schedules

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



=> Learning rate decay over time!

step decay:
e.g. decay learning rate by half every few epochs.

exponential decay:

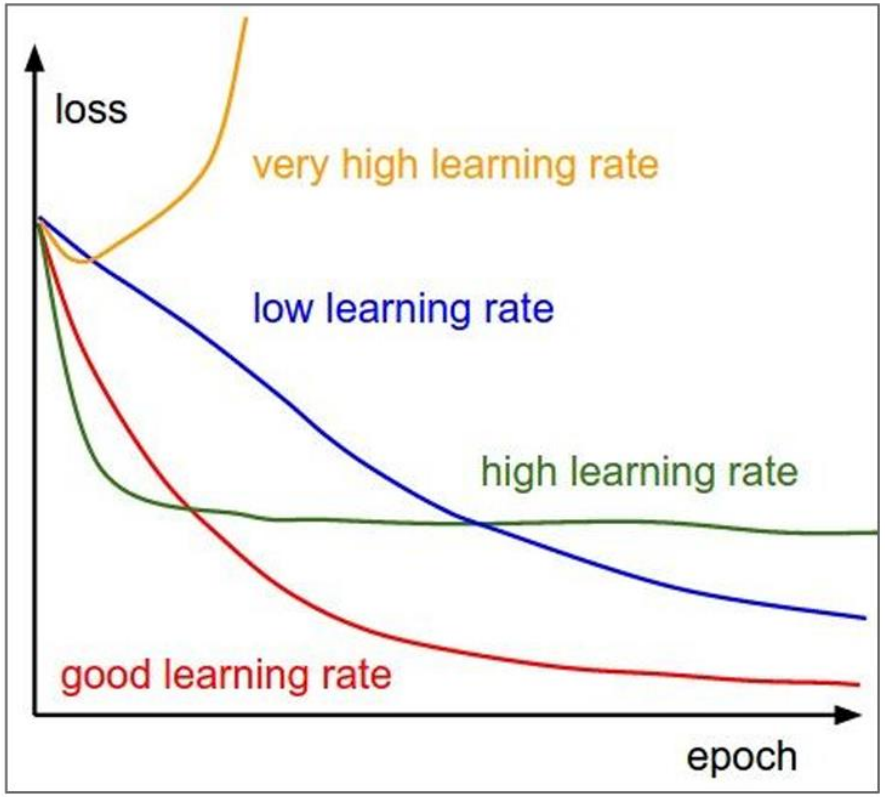
$$\alpha = \alpha_0 e^{-kt}$$

1/t decay:

$$\alpha = \alpha_0 / (1 + kt)$$

Learning Rate Schedules

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



=> Learning rate decay over time!

step decay:
e.g. decay learning rate by half every few epochs.

exponential decay:

$$\alpha = \alpha_0 e^{-kt}$$

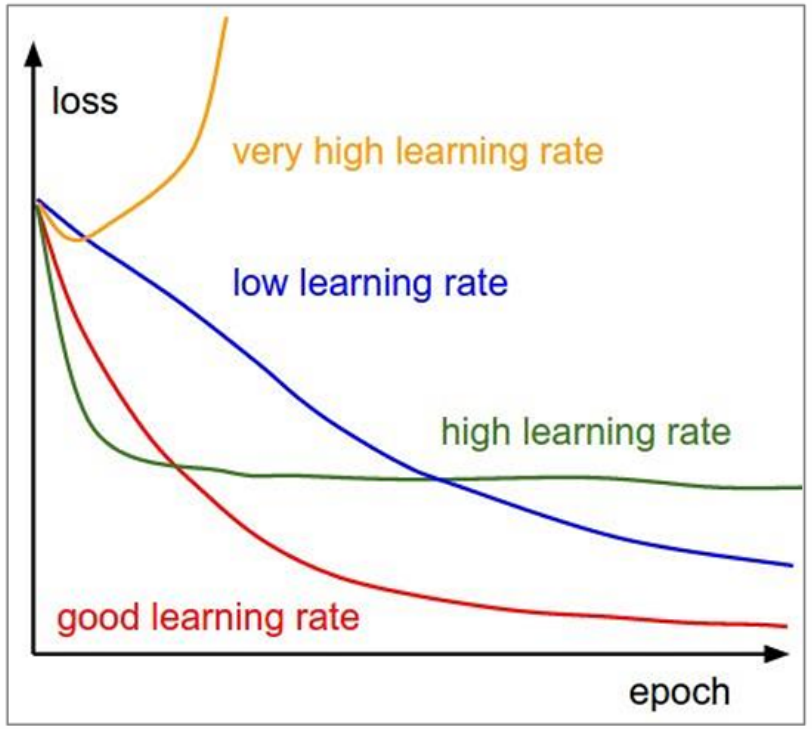
1/t decay:

$$\alpha = \alpha_0 / (1 + kt)$$

Not entirely true

Learning Rate Schedules

SGD, SGD+Momentum, RMSProp, Adam, AdamW all have **learning rate** as a hyperparameter.



Q: Which one of these learning rates is best to use?

A: In reality, all of these could be good learning rates.

In Practice

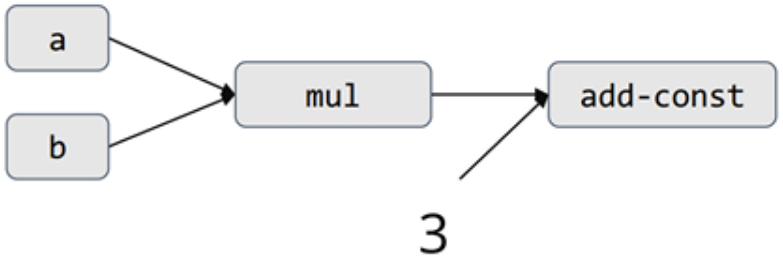
- **Adam(W)** is a good default choice in many cases; it often works ok even with constant learning rate
- **SGD+Momentum** can outperform Adam but may require more tuning of LR and schedule
- If you can afford to do full batch updates then look beyond 1st order optimization (**2nd order and beyond**)

Background: DL Computation

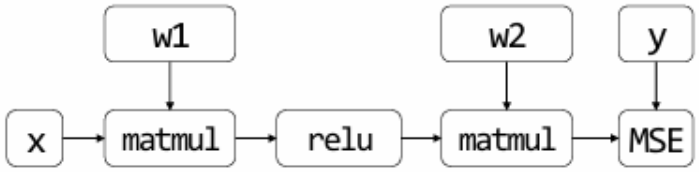
Dataflow graph representation

Computational Dataflow Graph

- Node: represents the computation (operator)
- Edge: represents the data dependency (data flowing direction)
- Node: also represents the *output tensor* of the operator
- Node: also represents an input constant tensor (if it is not an compute operator)



a x b + 3



$$L = \text{MSE}(w_2 \cdot \text{ReLU}(w_1 x),\ y)$$

Case Study: TensorFlow Program

- In the next few slides, we will do a case study of a deep learning program using TensorFlow v1 style API (classic Flavor).
- Note that today most deep learning frameworks now use a different style, but share the same mechanism under the hood
- Think about abstraction and implementation when going through these examples

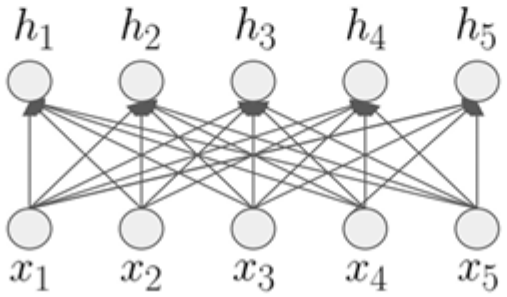
One linear NN: Logistic Regression

Input

$$x_i = \begin{bmatrix} \text{pixel}_1 \\ \text{pixel}_2 \\ \dots \\ \text{pixel}_m \end{bmatrix}$$

One Linear Layer

$$h_k = w_k^T x_i$$



Softmax

$$P(y_i = k|x_i) = \frac{\exp(h_k)}{\sum_{j=1}^{10} \exp(h_i)}$$

Whole Program

```
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

Loss Function

```python
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

Loss function **Declaration**

$$P(\text{label} = k) = y_k$$

$$L(y) = \sum_{k=1}^{10} I(\text{label} = k) \log(y_i)$$

Auto-Diff

```python
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

Automatic Differentiation

SGD Update

```python
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

SGD update rule

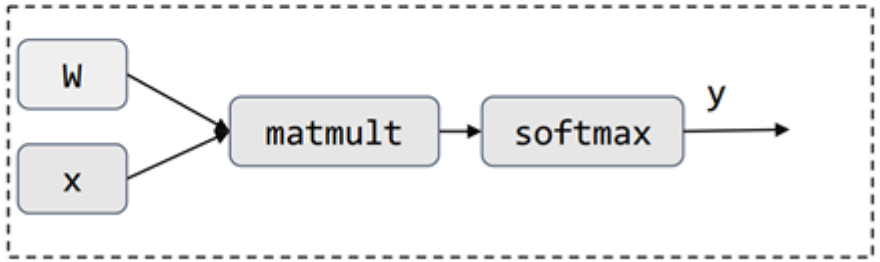
Trigger the Execution

```
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

Real execution happens here!

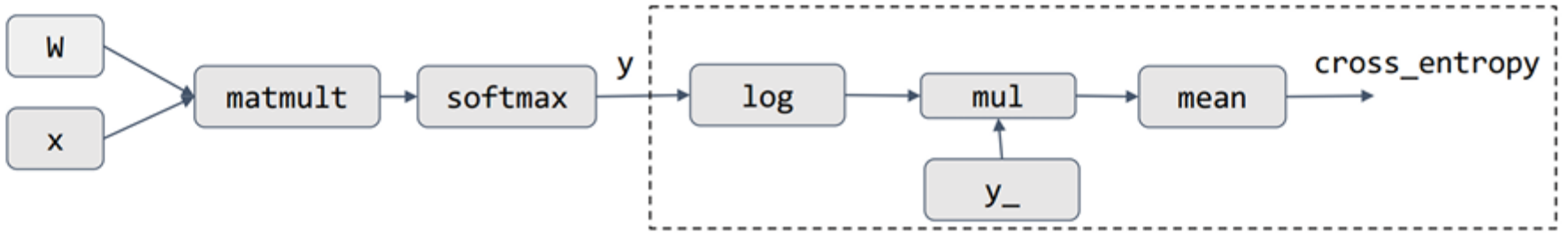
What happens behind the scene

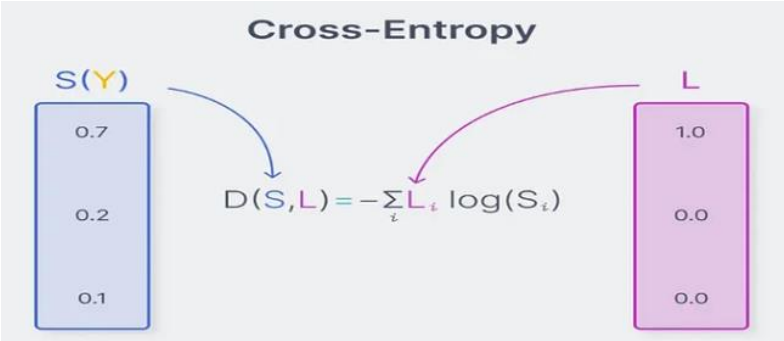
```
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
```



What happens behind the scene (Cond.)

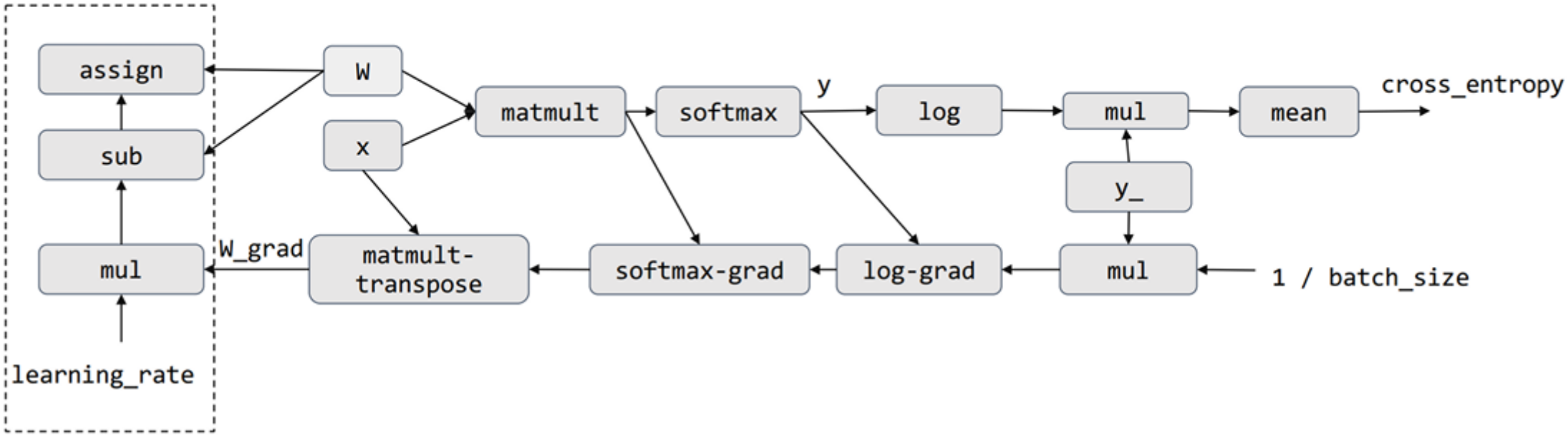
```
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
```





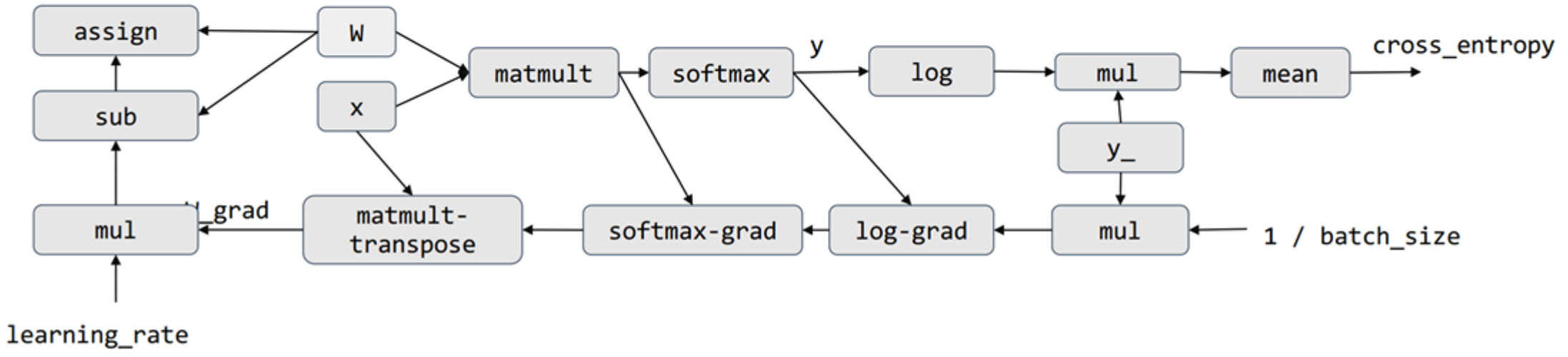
What happens behind the scene (Cond.)

```
sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```



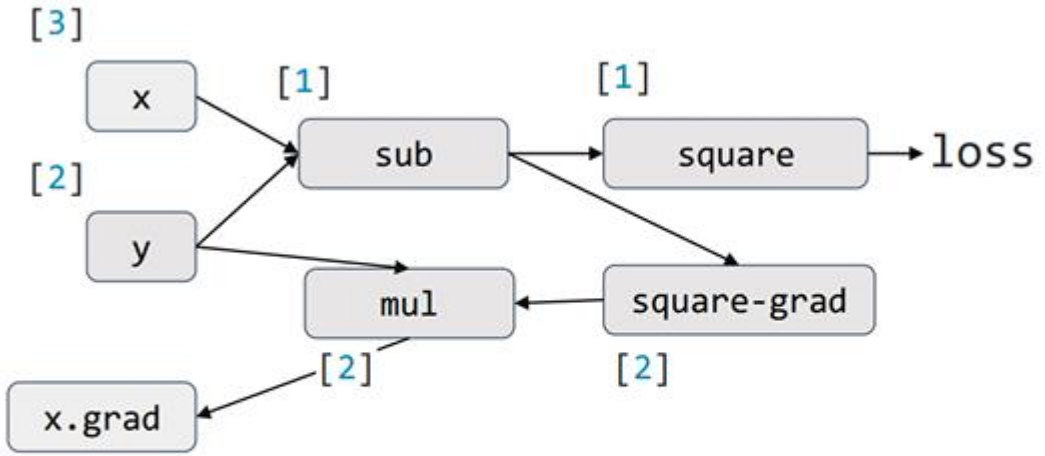
Discussion

- What are the benefits for computational graph abstraction?
- What are possible implementations and optimizations on this graph?
- What are the cons for computational graph abstraction?



A different flavor: PyTorch

```
x = torch.Tensor([3])
y = torch.Tensor([2])
z = x - y
loss = square(z)
loss.backward()
print(x.grad)
```



y.grad's path is omitted

Symbolic vs. Imperative

- Symbolic vs. imperative programming
- Define-then-run vs. define-and-run
- Define-then-run : write symbols to assemble the networks first, evaluate later
- define-and-run : immediate evaluation

```
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
```

Symbolic

```
x = torch.Tensor([3])
y = torch.Tensor([2])
z = x - y
loss = square(z)
loss.backward()
print(x.grad)
```

Imperative

Symbolic vs. Imperative

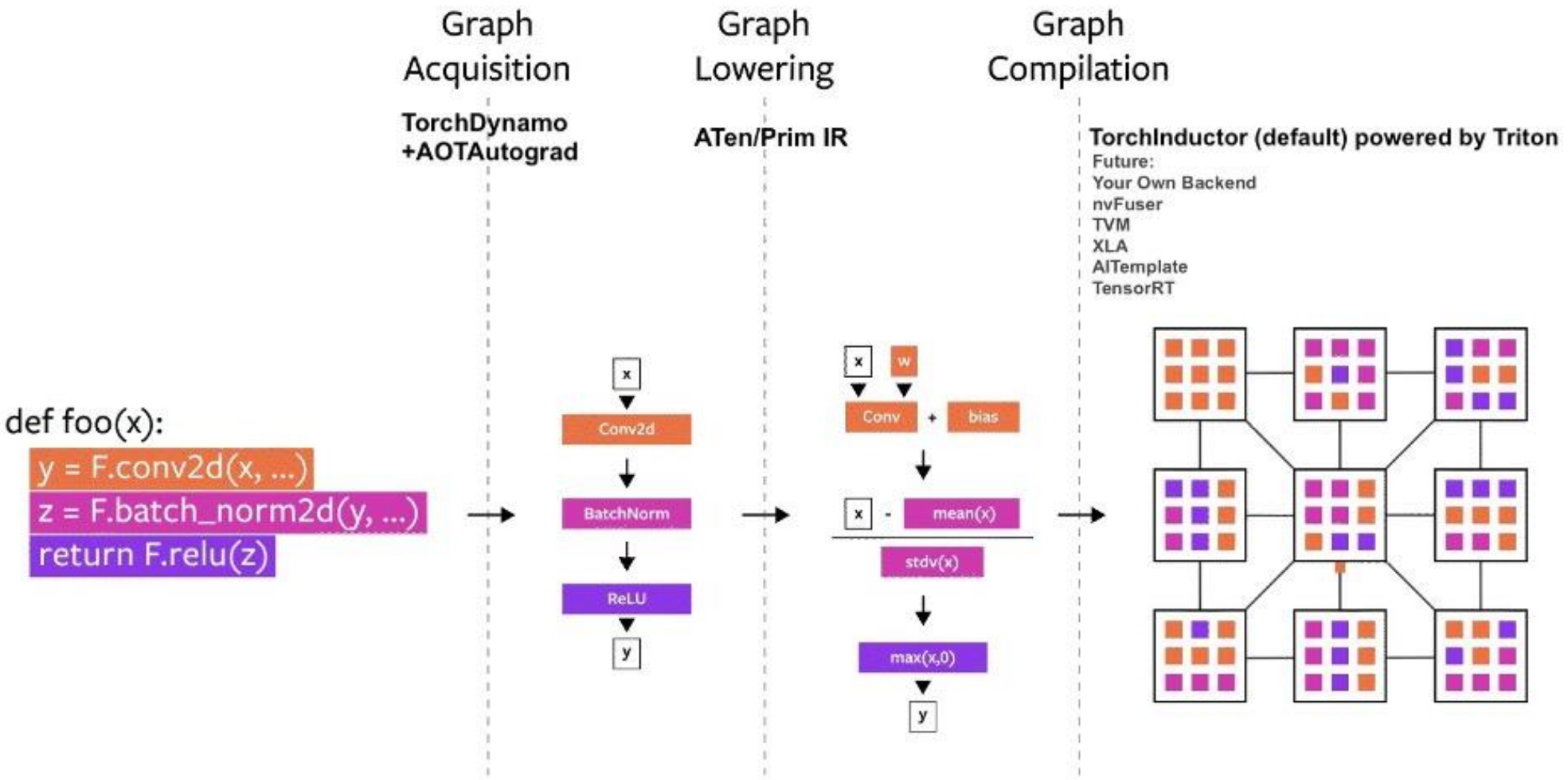
- Symbolic
  - Good
    - easy to optimize (e.g. distributed, batching, parallelization) for developers
    - More efficient
  - Bad
    - The way of programming might be counter-intuitive
    - Hard to debug for user programs
    - Less flexible: you need to write symbols before actually doing anything
- Imperative:
  - Good
    - More flexible: write one line, evaluate one line (that's why we all like Python)
    - Easy to program and easy to debug: because it matches the way we use C++ or python
  - Bad
    - Less efficient
    - More difficult to optimize

Just-in-time Compilation

- Ideally, we want define-and-run during _____
- We want define-then-run during _____
- Q: how can we have both without rewriting the program?

```
@torch.compile()
x = torch.Tensor([3])
y = torch.Tensor([2])
z = x - y
loss = square(z)
loss.backward()
print(x.grad)
```

PyTorch 2.0 Torch.compile



QA