DeepLearning.AI

Model Serialization and Version Control

Preparing Models for Deployment in PyTorch

Deployment

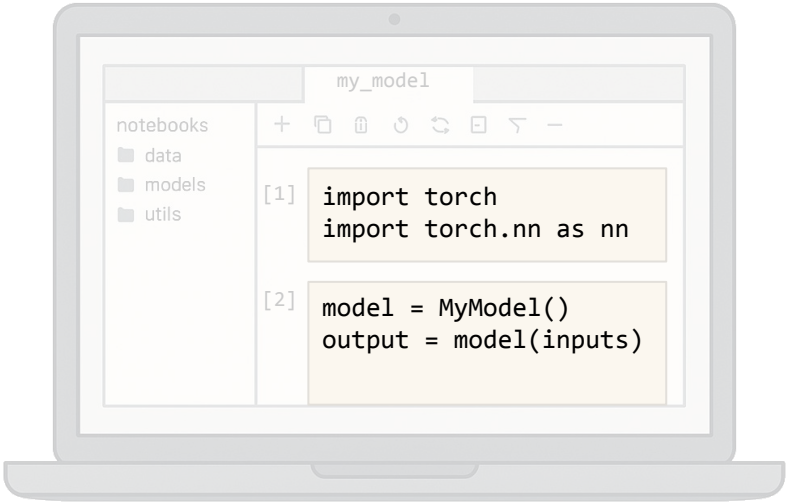
- **Save and Track**
- **Export**
- **Optimize**

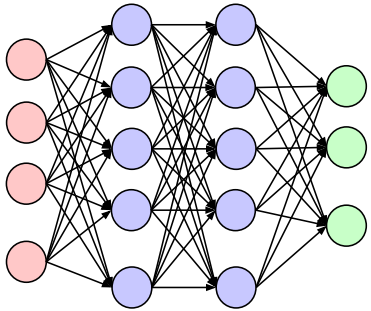
Deployment

- **Save and Track**
- Export
- Optimize

Serialization and Version Control

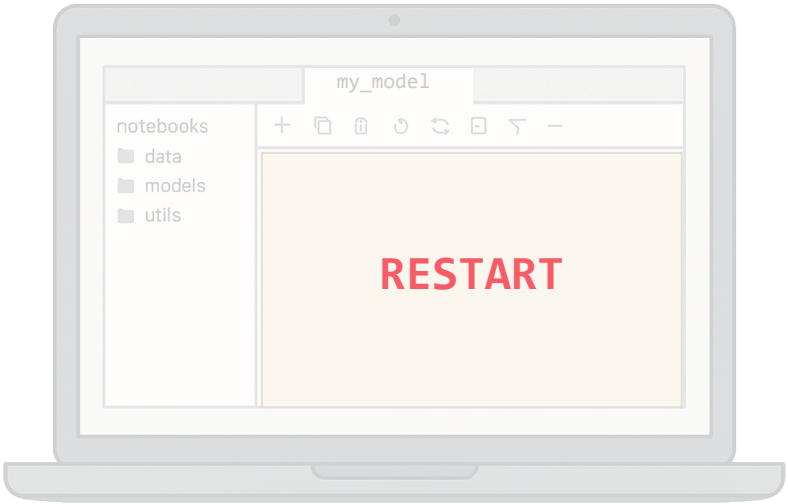
Saving and Tracking

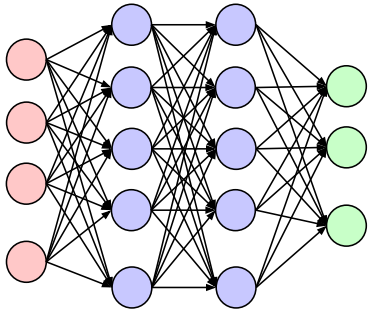




Trained Model

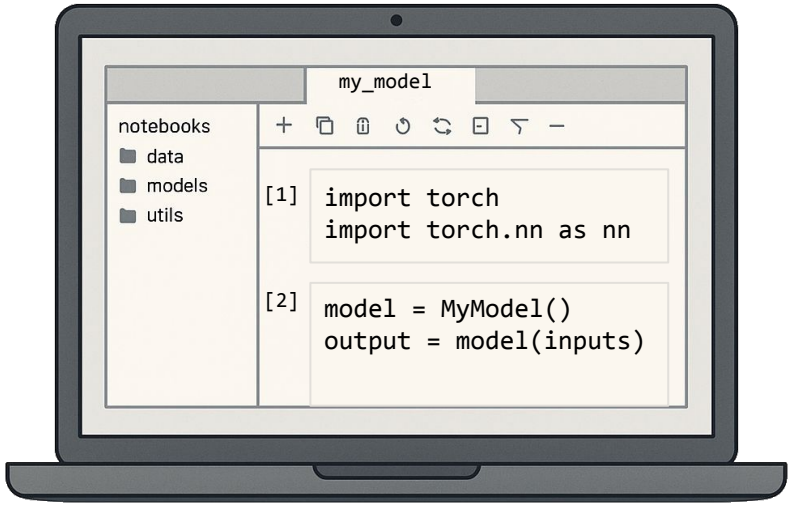
Saving and Tracking



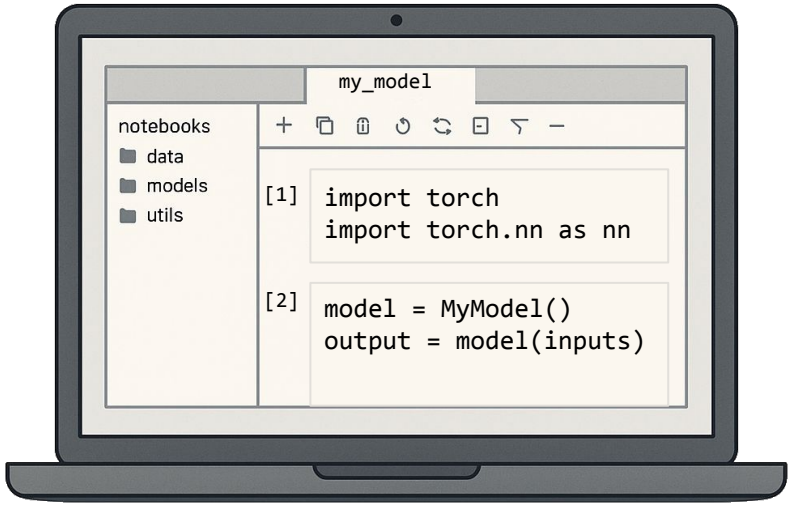


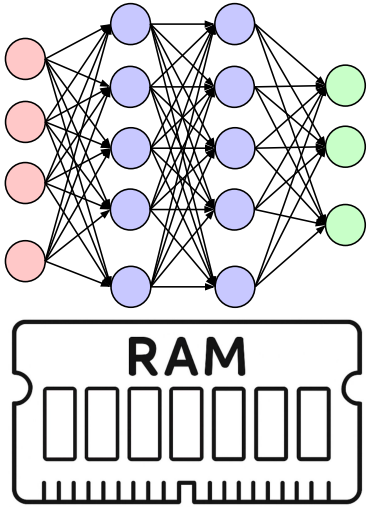
Trained Model

Saving and Tracking



Saving and Tracking





Serialization

```
model = SimpleCNN()
torch.save(model, 'my_model.pt')
```

Serialization

```
model = SimpleCNN()
torch.save(model, 'my_model.pt')
```

Serialization

```
model = SimpleCNN()
torch.save(model, 'my_model.pt')
```

Serialization

```
model = SimpleCNN()
torch.save(model, 'my_model.pt')
```

Serialization

```
model = SimpleCNN()
torch.save(model.state_dict(), 'my_trained_model.pt')
```

Serialization

```
model = SimpleCNN()
torch.save(model.state_dict(), 'my_trained_model.pt')
```

Serialization

```
model = SimpleCNN()
torch.save(model.state_dict(), 'my_trained_model.pt')

print(model.state_dict().keys())
```

```
odict_keys(['conv1.weight', 'conv1.bias', 'conv2.weight', 'conv2.bias', 'conv3.weight',
'conv3.bias', 'fc1.weight', 'fc1.bias', 'fc2.weight', 'fc2.bias'])
```

Serialization

```
model = SimpleCNN()
torch.save(model.state_dict(), 'my_trained_model.pt')

print(model.state_dict().keys())

model.load_state_dict(torch.load('my_trained_model.pt'))
```

```
odict_keys(['conv1.weight', 'conv1.bias', 'conv2.weight', 'conv2.bias', 'conv3.weight',
'conv3.bias', 'fc1.weight', 'fc1.bias', 'fc2.weight', 'fc2.bias'])
```

Serialization

```
model.load_state_dict(torch.load('downloaded_model.pt'))
```

Serialization

```
model.load_state_dict(torch.load('downloaded_model.pt'))
```

Serialization

```
model.load_state_dict(torch.load('downloaded_model.pt'))
```

Can execute arbitrary code!

Serialization

```
model.load_state_dict(torch.load('downloaded_model.pt', weights_only=True))
```

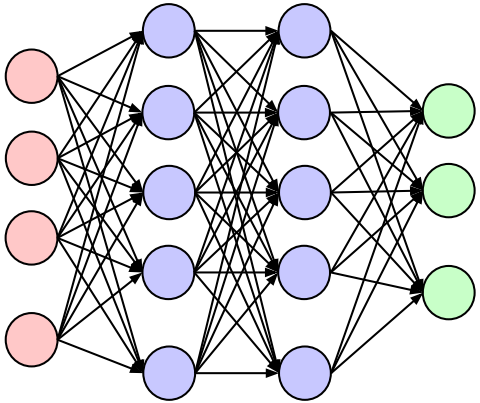
Serialization

```
model.load_state_dict(torch.load('downloaded_model.pt', weights_only=True))
```

Checkpoints

```
if epoch % 5 == 0:
    torch.save(checkpoint,
        f'checkpoint_epoch_{epoch}.pt')
```

Training...



Epoch 1

...

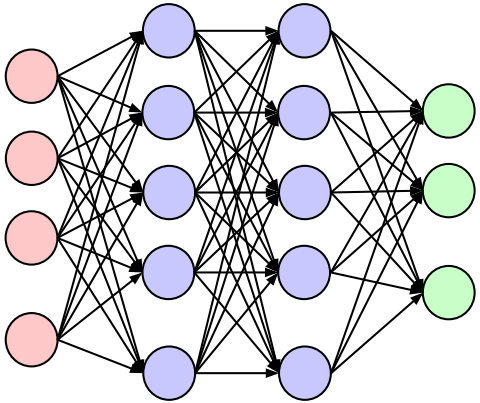
Epoch 5

Save!

Checkpoints

```
if epoch % 5 == 0:
    torch.save(checkpoint,
        f'checkpoint_epoch_{epoch}.pt')
```

Training...



Epoch 1

...

Epoch 5

Save!

Checkpoints

```
if epoch % 5 == 0:
    torch.save(checkpoint,
        f'checkpoint_epoch_{epoch}.pt')
```

Checkpoints

```
if epoch % 5 == 0:
    torch.save(checkpoint,
        f'checkpoint_epoch_{epoch}.pt')


checkpoint = {
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss,
}
```

Checkpoints

```
if epoch % 5 == 0:
    torch.save(checkpoint,
        f'checkpoint_epoch_{epoch}.pt')


checkpoint = {
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss,
}
```

Checkpoints

```
if epoch % 5 == 0:
    torch.save(checkpoint,
        f'checkpoint_epoch_{epoch}.pt')


checkpoint = {
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss,
}
```

Checkpoints

```
if epoch % 5 == 0:
    torch.save(checkpoint,
        f'checkpoint_epoch_{epoch}.pt')


checkpoint = {
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss,
}
```

Checkpoints

```
if epoch % 5 == 0:
    torch.save(checkpoint,
        f'checkpoint_epoch_{epoch}.pt')


checkpoint = {
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss,
}
```

Checkpoints

torch.save

checkpoint_epoch_5.pt
checkpoint_epoch_10.pt
checkpoint_epoch_15.pt

Checkpoints

torch.save

Model_final_final.pt
model_final_ACTUALLY_FINAL.pt
Model_final_use_this_one.pt

Metadata

```
torch.save(model.state_dict(), f'resnet18_acc{accuracy:.1f}_epoch{epoch}.pt')
```

Metadata

```
torch.save(model.state_dict(), f'resnet18_acc{accuracy:.1f}_epoch{epoch}.pt')
```

Metadata

```
checkpoint = {
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss,
    # Add any other training state you need
    'model_name': 'resnet18',
    'dataset_version': 'v2.1',
    'learning_rate': 0.001,
    'accuracy': accuracy,
    'date': datetime.now().isoformat()
    'notes': 'Training paused - accuracy still ~64%. Next: try lower LR or add dropout'
}
```

Metadata

```
checkpoint = {
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss,
    # Add any other training state you need
    'model_name': 'resnet18',
    'dataset_version': 'v2.1',
    'learning_rate': 0.001,
    'accuracy': accuracy,
    'date': datetime.now().isoformat()
    'notes': 'Training paused - accuracy still ~64%. Next: try lower LR or add dropout'
}
```

Metadata

```
checkpoint = {
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss,
    # Add any other training state you need
    'model_name': 'resnet18',
    'dataset_version': 'v2.1',
    'learning_rate': 0.001,
    'accuracy': accuracy,
    'date': datetime.now().isoformat()
    'notes': 'Training paused - accuracy still ~64%. Next: try lower LR or add dropout'
}
```

Metadata

```
checkpoint = {
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss,
    # Add any other training state you need
    'model_name': 'resnet18',
    'dataset_version': 'v2.1',
    'learning_rate': 0.001,
    'accuracy': accuracy,
    'date': datetime.now().isoformat()
    'notes': 'Training paused - accuracy still ~64%. Next: try lower LR or add dropout'
}
```

Metadata

```
checkpoint = {
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss,
    # Add any other training state you need
    'model_name': 'resnet18',
    'dataset_version': 'v2.1',
    'learning_rate': 0.001,
    'accuracy': accuracy,
    'date': datetime.now().isoformat()
    'notes': 'Training paused - accuracy still ~64%. Next: try lower LR or add dropout'
}
```

Metadata

```
checkpoint = {
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss,
    # Add any other training state you need
    'model_name': 'resnet18',
    'dataset_version': 'v2.1',
    'learning_rate': 0.001,
    'accuracy': accuracy,
    'date': datetime.now().isoformat()
    'notes': 'Training paused - accuracy still ~64%. Next: try lower LR or add dropout'
}
```

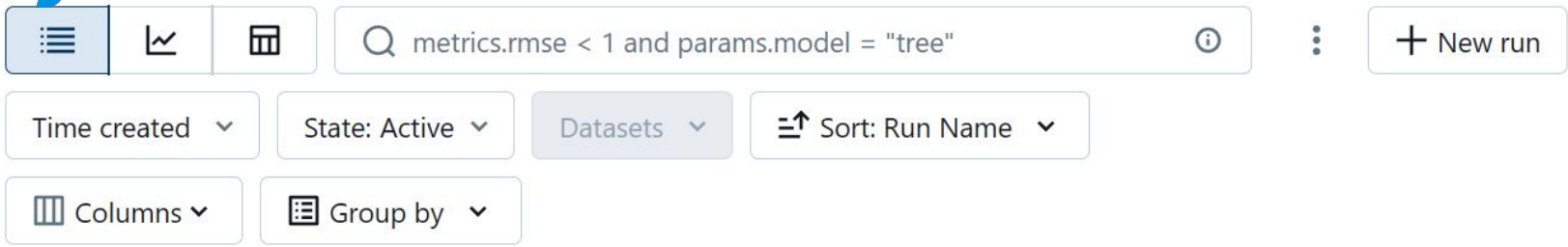
mlflow

mlflow

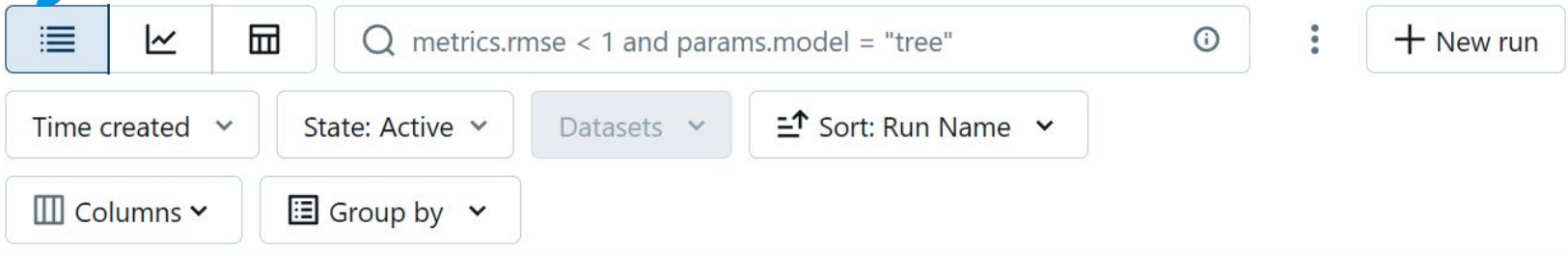
torch.save

Model_final_final.pt
model_final_ACTUALLY_FINAL.pt
Model_final_use_this_one.pt
chpt_v2b_final_maybe.pt
run17_overfit_szn.pt
tuesday_night_fix.pt
best_so_far_i_think.pt
dont_delete_this_one.pt
. . .

mlflow

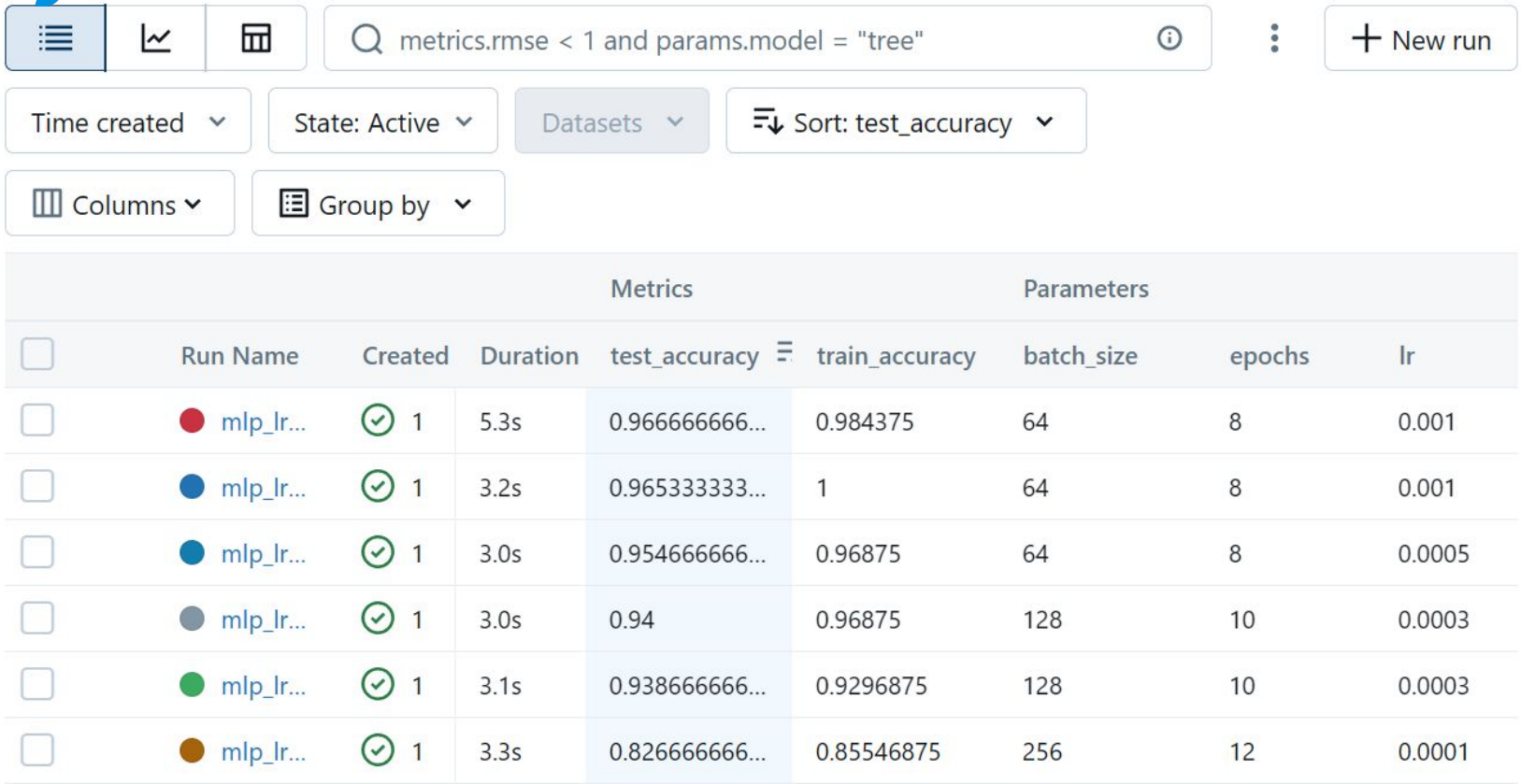

metrics.rmse < 1 and params.model = "tree"
New run
Time created
State: Active
Datasets
Sort: Run Name
Columns
Group by

mlflow



				Metrics		Parameters		
	Run Name	Created	Duration	test_accuracy	train_accuracy	batch_size	epochs	lr
	mlp_lr...	1	3.3s	0.826666666...	0.85546875	256	12	0.0001
	mlp_lr...	1	3.1s	0.938666666...	0.9296875	128	10	0.0003
	mlp_lr...	1	3.0s	0.94	0.96875	128	10	0.0003
	mlp_lr...	1	3.0s	0.954666666...	0.96875	64	8	0.0005
	mlp_lr...	1	5.3s	0.966666666...	0.984375	64	8	0.001
	mlp_lr...	1	3.2s	0.965333333...	1	64	8	0.001

mlflow



				Metrics		Parameters		
	Run Name	Created	Duration	test_accuracy	train_accuracy	batch_size	epochs	lr
	mlp_lr...	1	5.3s	0.966666666...	0.984375	64	8	0.001
	mlp_lr...	1	3.2s	0.965333333...	1	64	8	0.001
	mlp_lr...	1	3.0s	0.954666666...	0.96875	64	8	0.0005
	mlp_lr...	1	3.0s	0.94	0.96875	128	10	0.0003
	mlp_lr...	1	3.1s	0.938666666...	0.9296875	128	10	0.0003
	mlp_lr...	1	3.3s	0.826666666...	0.85546875	256	12	0.0001

mlflow

params.lr = "0.001"

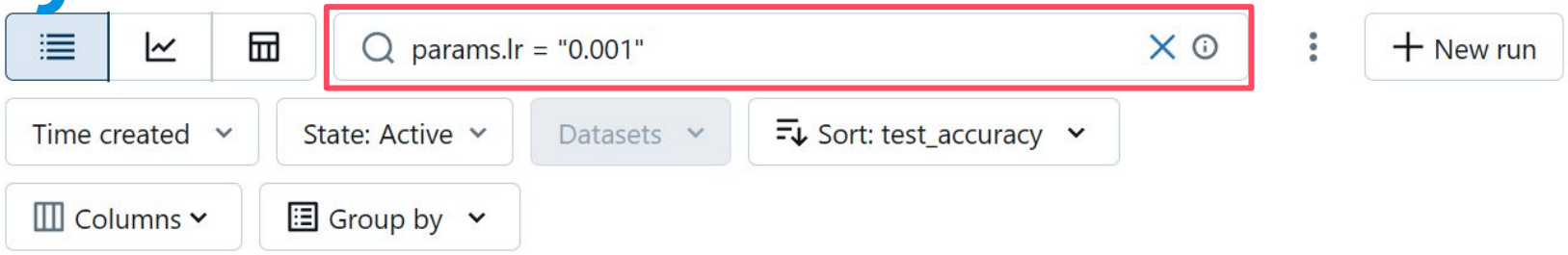
New run

Time created | State: Active | Datasets | Sort: test_accuracy

Columns | Group by

				Metrics		Parameters		
	Run Name	Created	Duration	test_accuracy	train_accuracy	batch_size	epochs	lr
	mlp_lr...	1	5.3s	0.966666666...	0.984375	64	8	0.001
	mlp_lr...	1	3.2s	0.965333333...	1	64	8	0.001
	mlp_lr...	1	3.0s	0.954666666...	0.96875	64	8	0.0005
	mlp_lr...	1	3.0s	0.94	0.96875	128	10	0.0003
	mlp_lr...	1	3.1s	0.938666666...	0.9296875	128	10	0.0003
	mlp_lr...	1	3.3s	0.826666666...	0.85546875	256	12	0.0001

mlflow



				Metrics		Parameters		
	Run Name	Created	Duration	test_accuracy	train_accuracy	batch_size	epochs	lr
	mlp_lr...	1	5.3s	0.966666666...	0.984375	64	8	0.001
	mlp_lr...	1	3.2s	0.965333333...	1	64	8	0.001

Comparing 5 runs

Parameters

Show diff only

batch_size	64	256	128	128	64
epochs	8	12	10	10	8
lr	0.0005	0.0001	0.0003	0.0003	0.001
seed	5	4	3	2	1
weight_decay	0.0005	0.0001	0.0	0.0001	0.001

Metrics

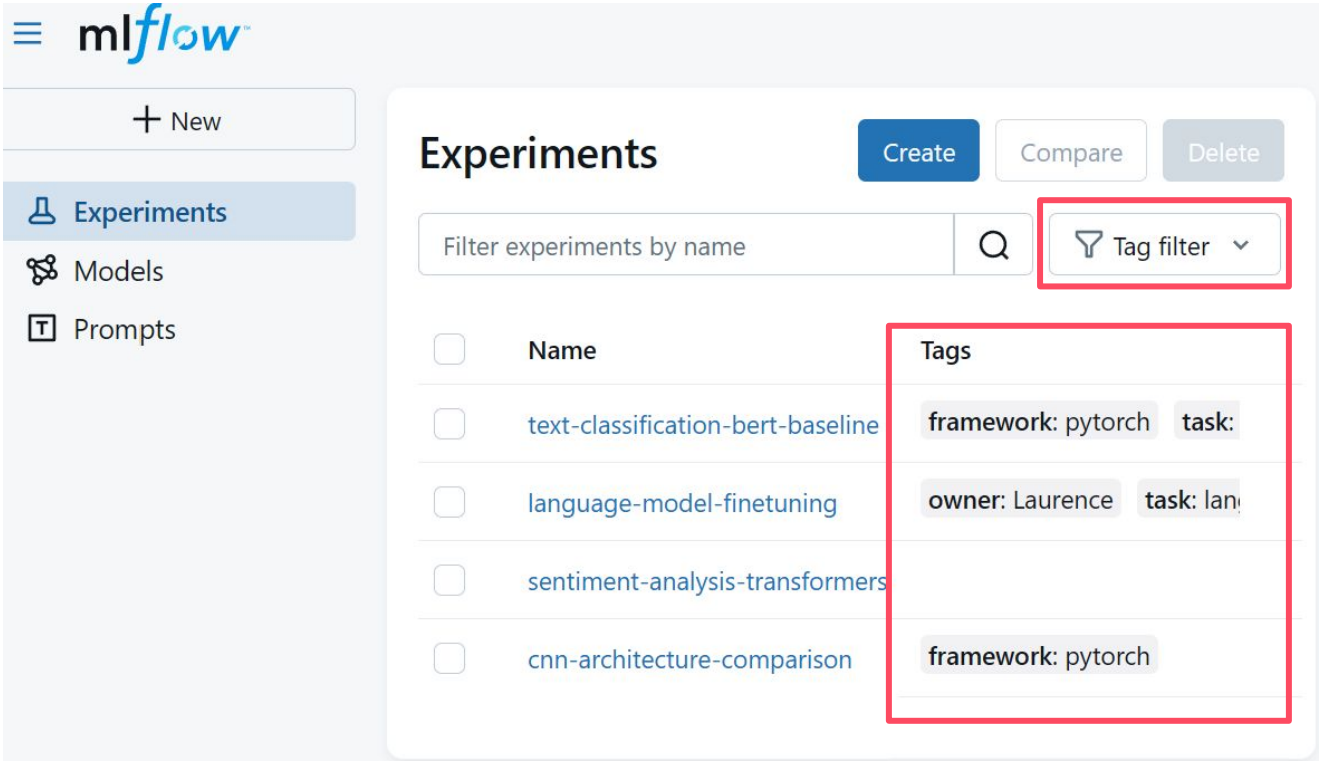
Show diff only

test_accuracy	0.955	0.827	0.94	0.939	0.965
train_accuracy	0.969	0.855	0.969	0.93	1
val_accuracy	0.953	0.832	0.923	0.932	0.961

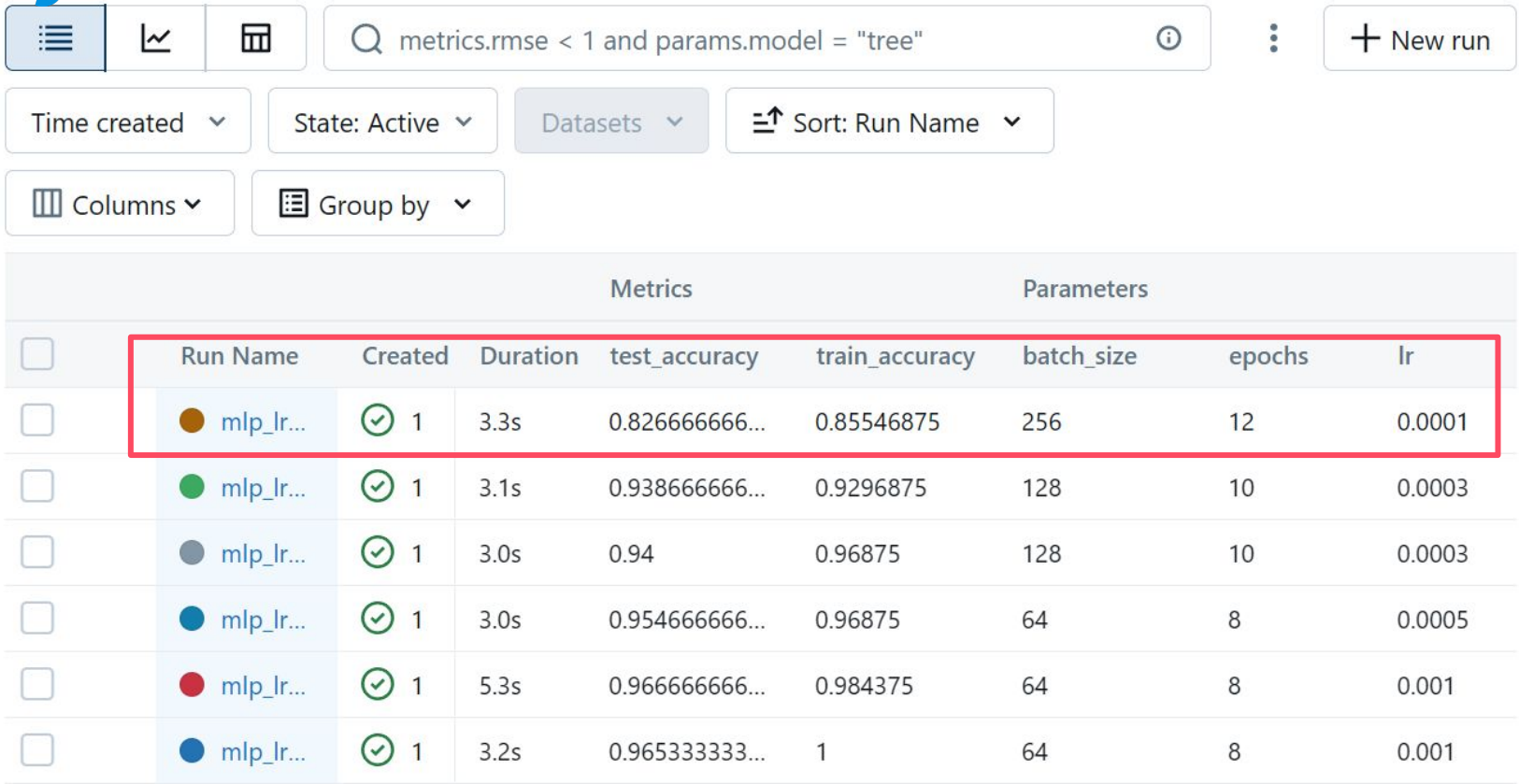
mlflow


image-clf-demo › Runs ›
mlp_lr=0.0005_wd=0.0005_bs=64_seed=5
Overview
Model metrics
System metrics
Traces
Artifacts
Search metric charts
Auto-refresh
Model metrics (3)
Add chart
test_accuracy
0.95
0
0.5
mlp_lr=0.0005_wd=0.0005_bs=64_seed=5
train_accuracy
0.95
0.9
0.85
0.8
2
4
6
8
Step
mlp_lr=0.0005_wd=0.0005_bs=64_seed=5 (trai...
val_accuracy
0.95
0.9
0.85
2
4
6
8
Step
mlp_lr=0.0005_wd=0.0005_bs=64_seed=5 (val_...

mlflow

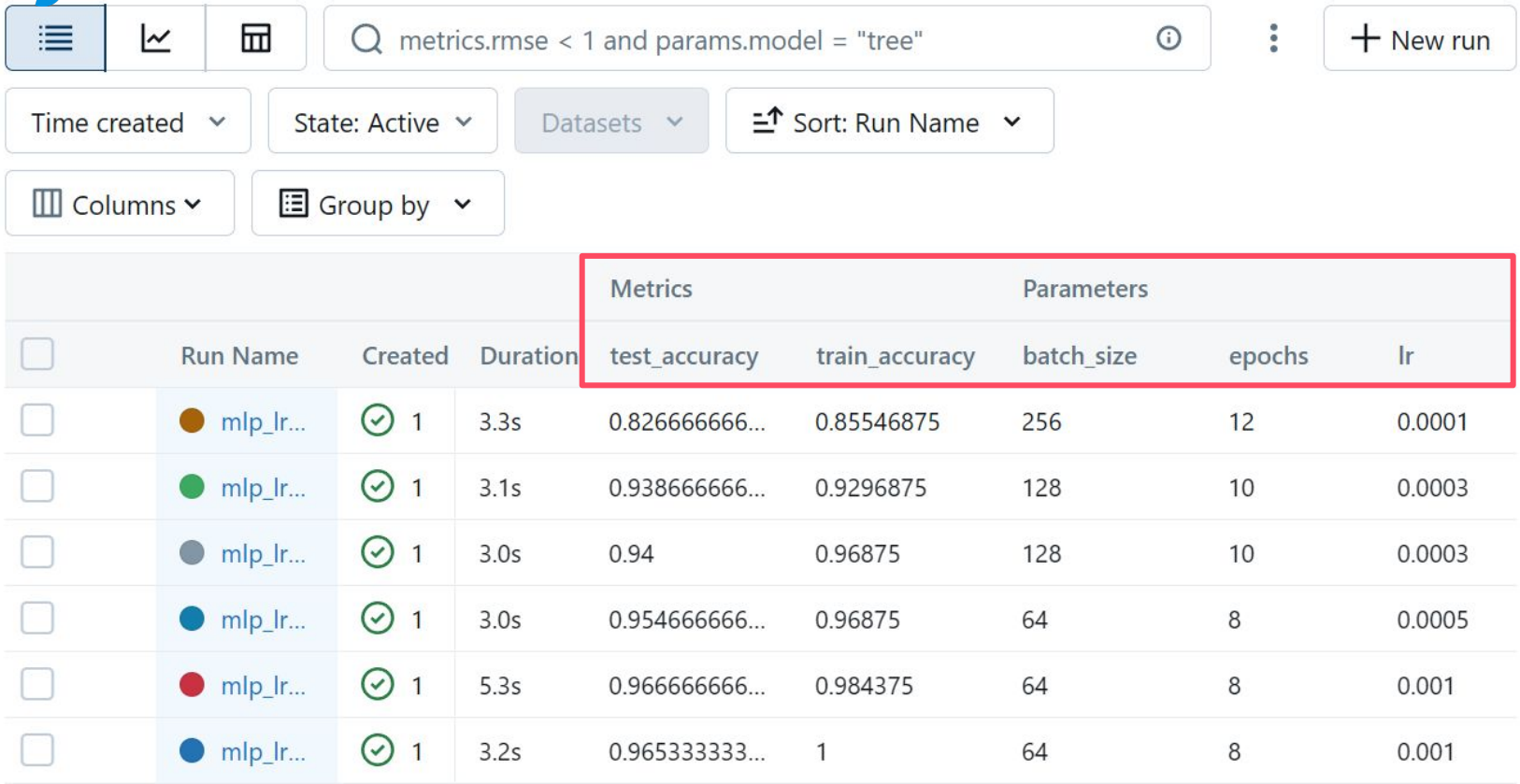

mlflow
New
Experiments
Models
Prompts
Experiments
Create
Compare
Delete
Filter experiments by name
Tag filter
Name
Tags
text-classification-bert-baseline
framework: pytorch
task:
language-model-finetuning
owner: Laurence
task: lan
sentiment-analysis-transformers
cnn-architecture-comparison
framework: pytorch

mlflow



				Metrics		Parameters		
	Run Name	Created	Duration	test_accuracy	train_accuracy	batch_size	epochs	lr
	mlp_lr...	1	3.3s	0.826666666...	0.85546875	256	12	0.0001
	mlp_lr...	1	3.1s	0.938666666...	0.9296875	128	10	0.0003
	mlp_lr...	1	3.0s	0.94	0.96875	128	10	0.0003
	mlp_lr...	1	3.0s	0.954666666...	0.96875	64	8	0.0005
	mlp_lr...	1	5.3s	0.966666666...	0.984375	64	8	0.001
	mlp_lr...	1	3.2s	0.965333333...	1	64	8	0.001

mlflow



				Metrics		Parameters		
	Run Name	Created	Duration	test_accuracy	train_accuracy	batch_size	epochs	lr
	mlp_lr...	1	3.3s	0.826666666...	0.85546875	256	12	0.0001
	mlp_lr...	1	3.1s	0.938666666...	0.9296875	128	10	0.0003
	mlp_lr...	1	3.0s	0.94	0.96875	128	10	0.0003
	mlp_lr...	1	3.0s	0.954666666...	0.96875	64	8	0.0005
	mlp_lr...	1	5.3s	0.966666666...	0.984375	64	8	0.001
	mlp_lr...	1	3.2s	0.965333333...	1	64	8	0.001

DeepLearning.AI

Exporting Models with ONNX

Preparing Models for Deployment in PyTorch

Deployment

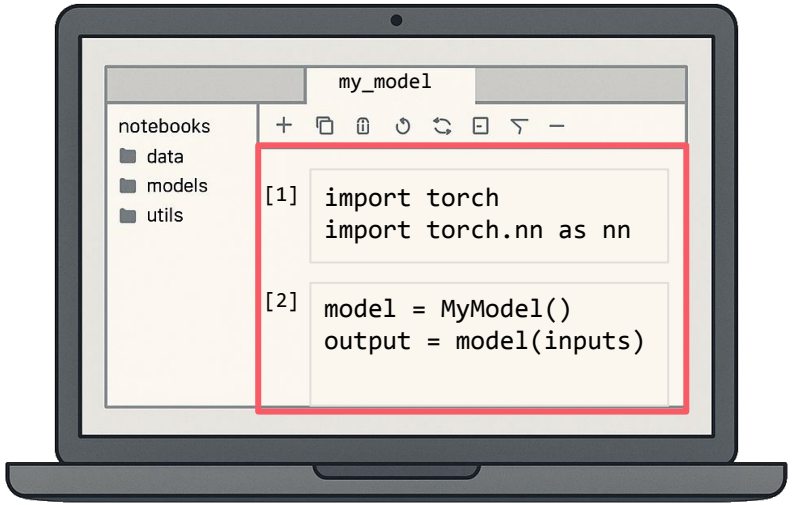
- **Save and Track**
- Export
- Optimize

Serialization and Version Control

Deployment

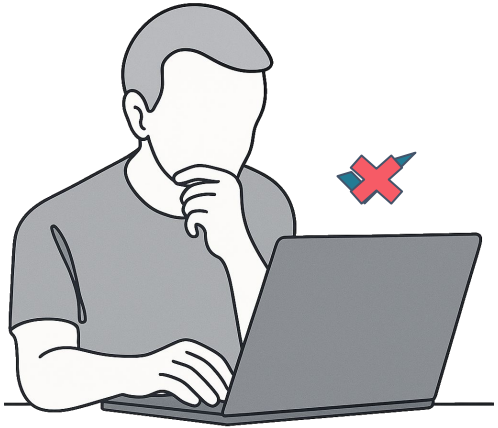
- Save and Track
- **Export**
- Optimize

Export

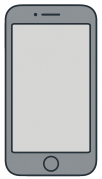


PyTorch-only

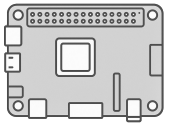
Export



servers



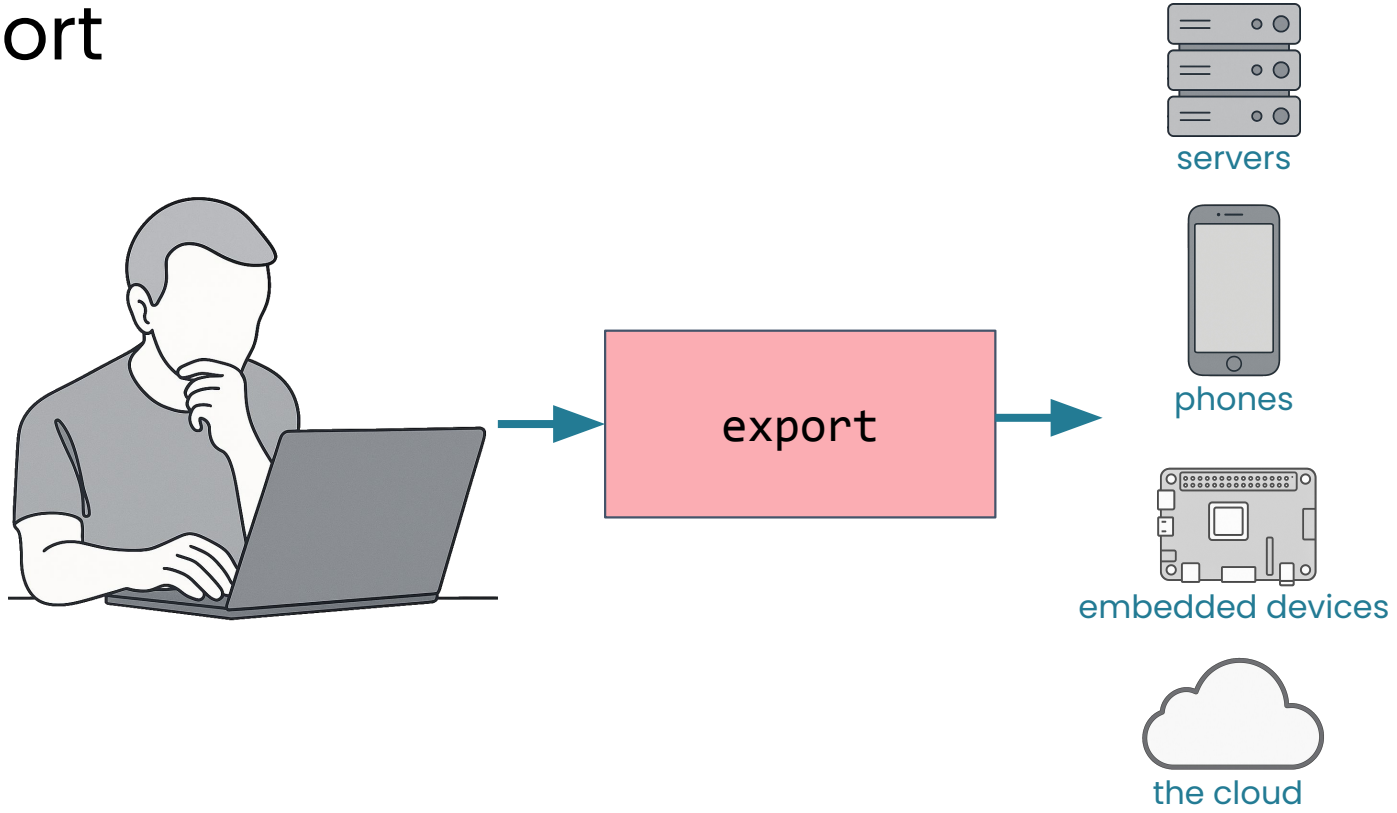
phones



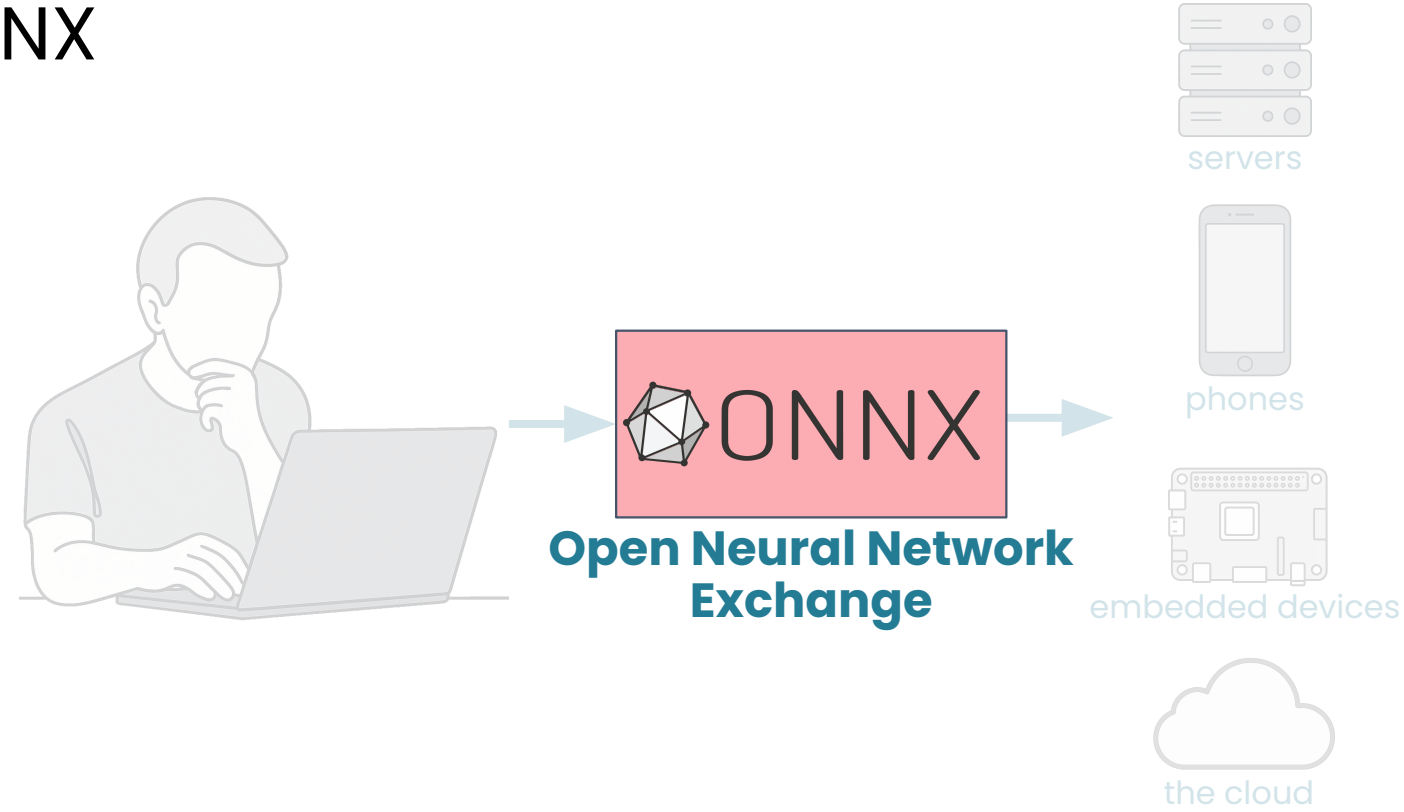
embedded devices

the cloud

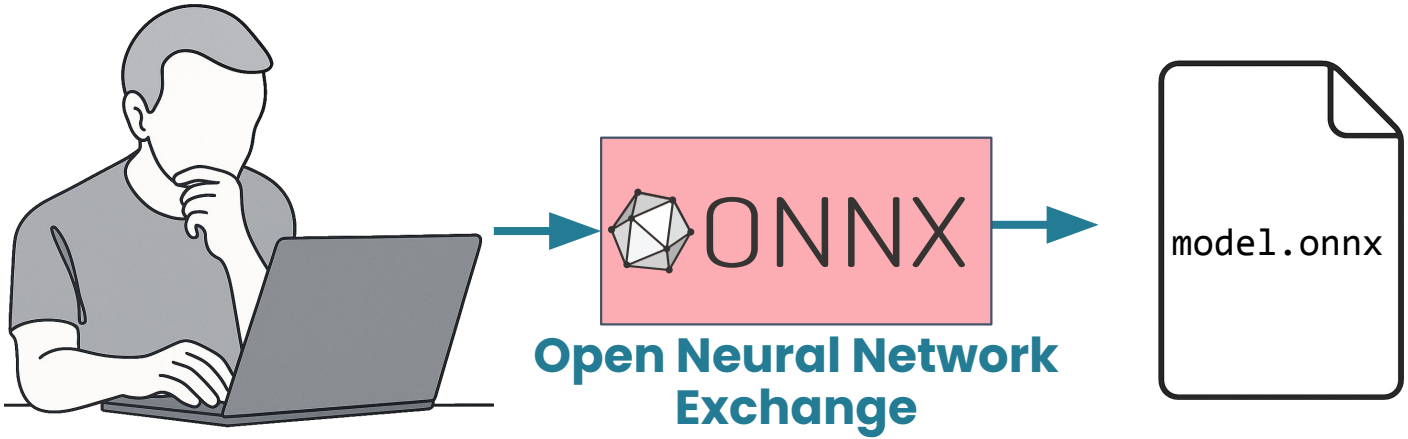
Export



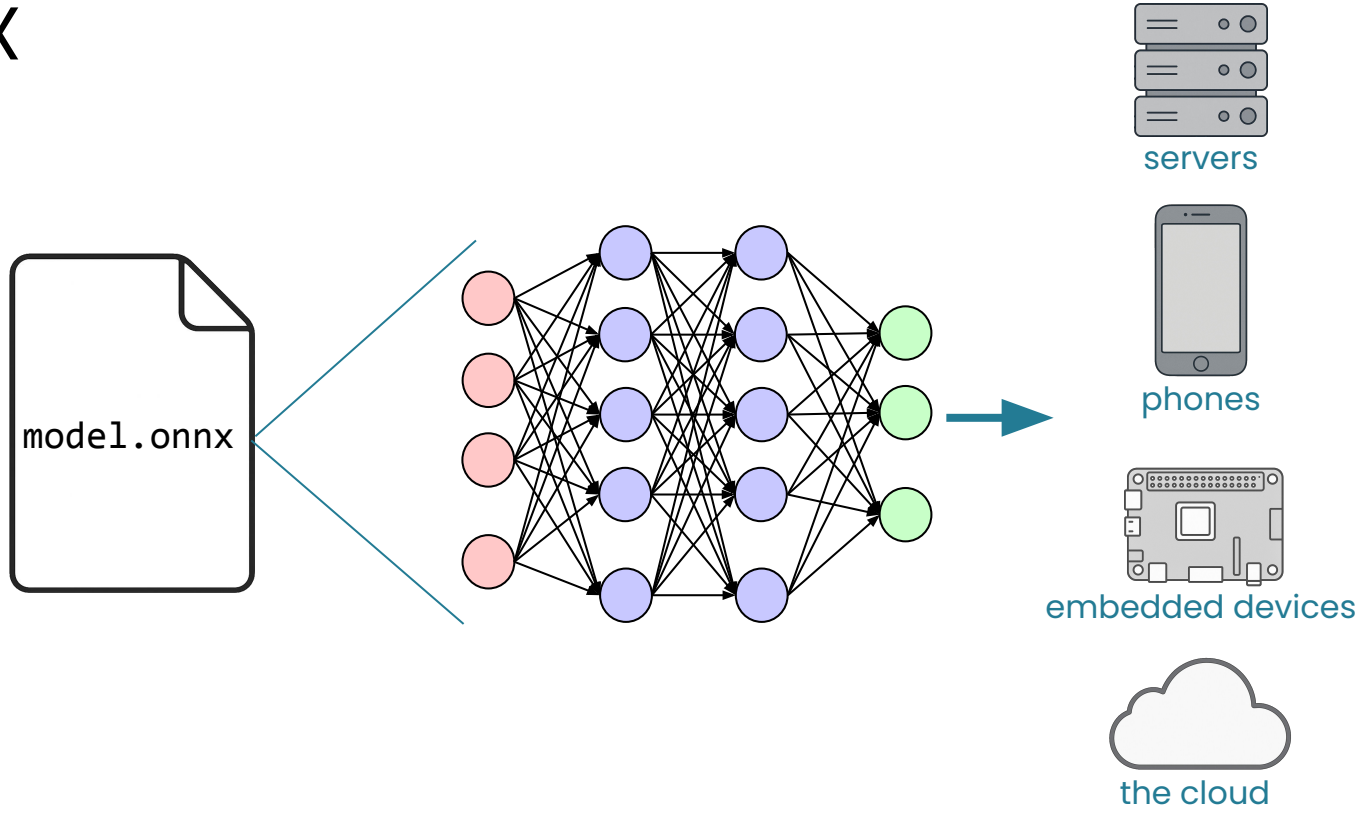
ONNX


servers
phones
ONNX
Open Neural Network Exchange
embedded devices
the cloud

ONNX



ONNX



Load a Model

```
model = MyModel()
model.load_state_dict(torch.load('model_weights.pt'))
model.eval()
```

Load a Model

```
model = MyModel()
model.load_state_dict(torch.load('model_weights.pt'))
model.eval()
```

Load a Model

```
model = MyModel()
model.load_state_dict(torch.load('model_weights.pt'))
model.eval()
```

Load a Model

```
model = MyModel()
model.load_state_dict(torch.load('model_weights.pt'))
model.eval()
```

Load a Model

```
model = MyModel()
model.load_state_dict(torch.load('model_weights.pt'))
model.eval()

# Example tensor with expected input shape
dummy_input = torch.randn(1, 3, 224, 224)
```

Load a Model

```
model = MyModel()
model.load_state_dict(torch.load('model_weights.pt'))
model.eval()

# Example tensor with expected input shape
dummy_input = torch.randn(1, 3, 224, 224)
```

Load a Model

```
model = MyModel()
model.load_state_dict(torch.load('model_weights.pt'))
model.eval()

# Example tensor with expected input shape
dummy_input = torch.randn(1, 3, 224, 224)
```

batch size

Load a Model

```
model = MyModel()
model.load_state_dict(torch.load('model_weights.pt'))
model.eval()

# Example tensor with expected input shape
dummy_input = torch.randn(1, 3, 224, 224)
```

RGB channels

Load a Model

```
model = MyModel()
model.load_state_dict(torch.load('model_weights.pt'))
model.eval()

# Example tensor with expected input shape
dummy_input = torch.randn(1, 3, 224, 224)
```

height & width

Export to ONNX

```
# Export to ONNX format
torch.onnx.export(
    model,                  # PyTorch model
    dummy_input,            # Input tensor
    "model.onnx",           # Output file name
    export_params=True,     # Store weights in model file
    opset_version=11,       # ONNX version
    do_constant_folding=True,  # Optimize constant folding
    input_names=['input'],      # Name inputs
    output_names=['output'],    # Name outputs
    dynamic_axes={              # Support variable batch size
        'input': {0: 'batch_size'},
        'output': {0: 'batch_size'}
    }
)
```

Export to ONNX

```
# Export to ONNX format
torch.onnx.export(
    model,                  # PyTorch model
    dummy_input,            # Input tensor
    "model.onnx",           # Output file name
    export_params=True,     # Store weights in model file
    opset_version=11,       # ONNX version
    do_constant_folding=True,  # Optimize constant folding
    input_names=['input'],      # Name inputs
    output_names=['output'],    # Name outputs
    dynamic_axes={              # Support variable batch size
        'input': {0: 'batch_size'},
        'output': {0: 'batch_size'}
    }
)
```

Export to ONNX

```
# Export to ONNX format
torch.onnx.export(
    model,                  # PyTorch model
    dummy_input,            # Input tensor
    "model.onnx",           # Output file name
    export_params=True,     # Store weights in model file
    opset_version=11,       # ONNX version
    do_constant_folding=True,  # Optimize constant folding
    input_names=['input'],      # Name inputs
    output_names=['output'],    # Name outputs
    dynamic_axes={              # Support variable batch size
        'input': {0: 'batch_size'},
        'output': {0: 'batch_size'}
    }
)
```

Export to ONNX

```
# Export to ONNX format
torch.onnx.export(
    model,                  # PyTorch model
    dummy_input,            # Input tensor
    "model.onnx",           # Output file name
    export_params=True,     # Store weights in model file
    opset_version=11,       # ONNX version
    do_constant_folding=True,  # Optimize constant folding
    input_names=['input'],      # Name inputs
    output_names=['output'],    # Name outputs
    dynamic_axes={              # Support variable batch size
        'input': {0: 'batch_size'},
        'output': {0: 'batch_size'}
    }
)
```

Export to ONNX

```
# Export to ONNX format
torch.onnx.export(
    model,                  # PyTorch model
    dummy_input,            # Input tensor
    "model.onnx",           # Output file name
    export_params=True,     # Store weights in model file
    opset_version=11,       # ONNX version
    do_constant_folding=True,  # Optimize constant folding
    input_names=['input'],      # Name inputs
    output_names=['output'],    # Name outputs
    dynamic_axes={              # Support variable batch size
        'input': {0: 'batch_size'},
        'output': {0: 'batch_size'}
    }
)
```

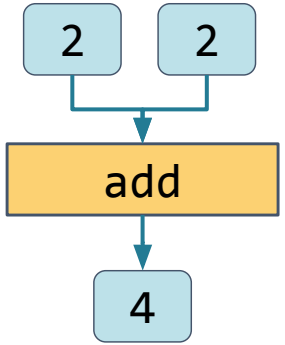
Export to ONNX

```
# Export to ONNX format
torch.onnx.export(
    model,                  # PyTorch model
    dummy_input,            # Input tensor
    "model.onnx",           # Output file name
    export_params=True,     # Store weights in model file
    opset_version=11,       # ONNX version
    do_constant_folding=True,  # Optimize constant folding
    input_names=['input'],      # Name inputs
    output_names=['output'],    # Name outputs
    dynamic_axes={              # Support variable batch size
        'input': {0: 'batch_size'},
        'output': {0: 'batch_size'}
    }
)
```

Operator	Since Version
AveragePool	22,19,11,10..
Flatten	24,23,21,13..
MatMul	13,9,1
MaxPool	22,12,11,10..
Reshape	24,23,21,19..
Squeeze	24,23,21,13..

Export to ONNX

```
# Export to ONNX format
torch.onnx.export(
    model,                  # PyTorch model
    dummy_input,            # Input tensor
    "model.onnx",           # Output file name
    export_params=True,     # Store weights in model file
    opset_version=11,       # ONNX version
    do_constant_folding=True,  # Optimize constant folding
    input_names=['input'],      # Name inputs
    output_names=['output'],    # Name outputs
    dynamic_axes={              # Support variable batch size
        'input': {0: 'batch_size'},
        'output': {0: 'batch_size'}
    }
)
```



Export to ONNX

```
# Export to ONNX format
torch.onnx.export(
    model,                  # PyTorch model
    dummy_input,            # Input tensor
    "model.onnx",           # Output file name
    export_params=True,     # Store weights in model file
    opset_version=11,       # ONNX version
    do_constant_folding=True,  # Optimize constant folding
    input_names=['input'],      # Name inputs
    output_names=['output'],    # Name outputs
    dynamic_axes={              # Support variable batch size
        'input': {0: 'batch_size'},
        'output': {0: 'batch_size'}
    }
)
```

4

Export to ONNX

```
# Export to ONNX format
torch.onnx.export(
    model,                  # PyTorch model
    dummy_input,            # Input tensor
    "model.onnx",           # Output file name
    export_params=True,     # Store weights in model file
    opset_version=11,       # ONNX version
    do_constant_folding=True,  # Optimize constant folding
    input_names=['input'],      # Name inputs
    output_names=['output'],    # Name outputs
    dynamic_axes={              # Support variable batch size
        'input': {0: 'batch_size'},
        'output': {0: 'batch_size'}
    }
)
```

Export to ONNX

```
# Export to ONNX format
torch.onnx.export(
    model,                  # PyTorch model
    dummy_input,            # Input tensor
    "model.onnx",           # Output file name
    export_params=True,     # Store weights in model file
    opset_version=11,       # ONNX version
    do_constant_folding=True,  # Optimize constant folding
    input_names=['images'],         # Name inputs
    output_names=['predictions'],   # Name outputs
    dynamic_axes={              # Support variable batch size
        'input': {0: 'batch_size'},
        'output': {0: 'batch_size'}
    }
)
```

Export to ONNX

```
# Export to ONNX format
torch.onnx.export(
    model,                  # PyTorch model
    dummy_input,            # Input tensor
    "model.onnx",           # Output file name
    export_params=True,     # Store weights in model file
    opset_version=11,       # ONNX version
    do_constant_folding=True,  # Optimize constant folding
    input_names=['images'],        # Name inputs
    output_names=['predictions'],  # Name outputs
    dynamic_axes={
        'input': {0: 'batch_size'},
        'output': {0: 'batch_size'}
    }
)
```

```
torch.randn(1, 3, 224, 224)
```

Export to ONNX

```
# Export to ONNX format
torch.onnx.export(
    model,                  # PyTorch model
    dummy_input,            # Input tensor
    "model.onnx",           # Output file name
    export_params=True,     # Store weights in model file
    opset_version=11,       # ONNX version
    do_constant_folding=True,  # Optimize constant folding
    input_names=['images'],        # Name inputs
    output_names=['predictions'],  # Name outputs
    dynamic_axes={
        'input': {0: 'batch_size'},
        'output': {0: 'batch_size'}
    }
)
```

```
torch.randn(1, 3, 224, 224)
```

ONNX Inference

```
import onnxruntime as ort
import numpy as np

# Load ONNX model
session = ort.InferenceSession("model.onnx")

# Prepare input
input_name = session.get_inputs()[0].name
input_data = np.random.randn(1, 3, 224, 224).astype(np.float32)

# Run inference
outputs = session.run(None, {input_name: input_data})
prediction = outputs[0]
```

ONNX Inference

```
import onnxruntime as ort
import numpy as np

# Load ONNX model
session = ort.InferenceSession("model.onnx")

# Prepare input
input_name = session.get_inputs()[0].name
input_data = np.random.randn(1, 3, 224, 224).astype(np.float32)

# Run inference
outputs = session.run(None, {input_name: input_data})
prediction = outputs[0]
```

ONNX Inference

```
import onnxruntime as ort
import numpy as np

# Load ONNX model
session = ort.InferenceSession("model.onnx")

# Prepare input
input_name = session.get_inputs()[0].name
input_data = np.random.randn(1, 3, 224, 224).astype(np.float32)

# Run inference
outputs = session.run(None, {input_name: input_data})
prediction = outputs[0]
```

ONNX Inference

```
import onnxruntime as ort
import numpy as np

# Load ONNX model
session = ort.InferenceSession("model.onnx")

# Prepare input
input_name = session.get_inputs()[0].name
input_data = np.random.randn(1, 3, 224, 224).astype(np.float32)

# Run inference
outputs = session.run(None, {input_name: input_data})
prediction = outputs[0]
```

ONNX Inference

```
import onnxruntime as ort
import numpy as np

# Load ONNX model
session = ort.InferenceSession("model.onnx")

# Prepare input
input_name = session.get_inputs()[0].name
input_data = np.random.randn(1, 3, 224, 224).astype(np.float32)

# Run inference
outputs = session.run(None, {input_name: input_data})
prediction = outputs[0]
```

ONNX Inference

```
import onnxruntime as ort
import numpy as np

# Load ONNX model
session = ort.InferenceSession("model.onnx")

# Prepare input
input_name = session.get_inputs()[0].name
input_data = np.random.randn(1, 3, 224, 224).astype(np.float32)

# Run inference
outputs = session.run(None, {input_name: input_data})
prediction = outputs[0]
```

ONNX Inference

```
import onnxruntime as ort
import numpy as np

# Load ONNX model
session = ort.InferenceSession("model.onnx")

# Prepare input
input_name = session.get_inputs()[0].name
input_data = np.random.randn(1, 3, 224, 224).astype(np.float32)

# Run inference
outputs = session.run(None, {input_name: input_data})
prediction = outputs[0]
```

ONNX Inference

```
import onnxruntime as ort
import numpy as np

# Load ONNX model
session = ort.InferenceSession("model.onnx")

# Prepare input
input_name = session.get_inputs()[0].name
input_data = np.random.randn(1, 3, 224, 224).astype(np.float32)

# Run inference
outputs = session.run(None, {input_name: input_data})
prediction = outputs[0]
```

ONNX Inference

```
import onnxruntime as ort
import numpy as np

# Load ONNX model
session = ort.InferenceSession("model.onnx")

# Prepare input
input_name = session.get_inputs()[0].name
input_data = np.random.randn(1, 3, 224, 224).astype(np.float32)

# Run inference
outputs = session.run(None, {input_name: input_data})
prediction = outputs[0]
```

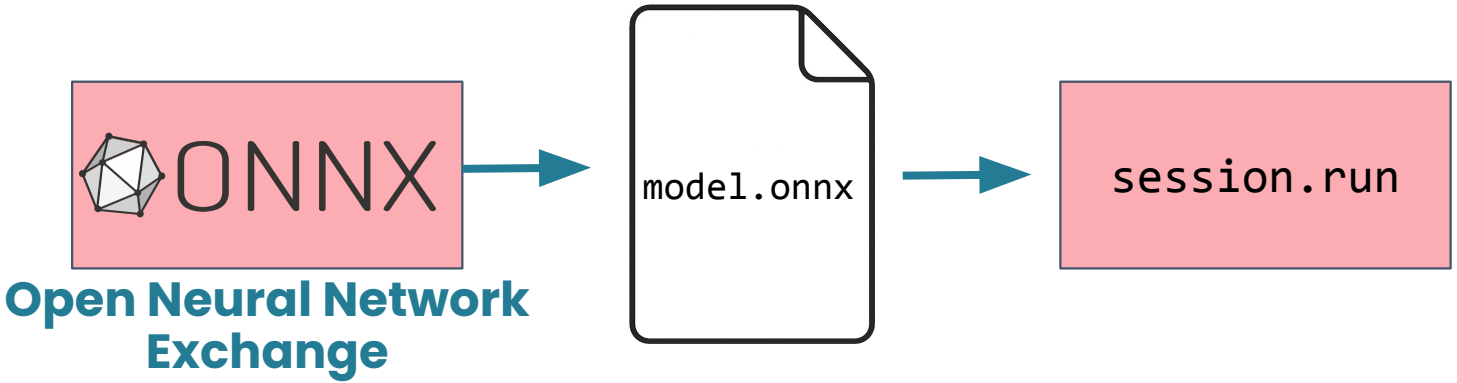
ONNX Inference

```
import onnxruntime as ort
import numpy as np

# Load ONNX model
session = ort.InferenceSession("model.onnx")

# Prepare input
input_name = session.get_inputs()[0].name
input_data = np.random.randn(1, 3, 224, 224).astype(np.float32)

# Run inference
outputs = session.run(None, {input_name: input_data})
prediction = outputs[0]
```

ONNX Inference

```
import onnxruntime as ort
import numpy as np

# Load ONNX model
session = ort.InferenceSession("model.onnx")

# Prepare input
input_name = session.get_inputs()[0].name
input_data = np.random.randn(1, 3, 224, 224).astype(np.float32)

# Run inference
outputs = session.run(None, {input_name: input_data})
prediction = outputs[0]
```

ONNX

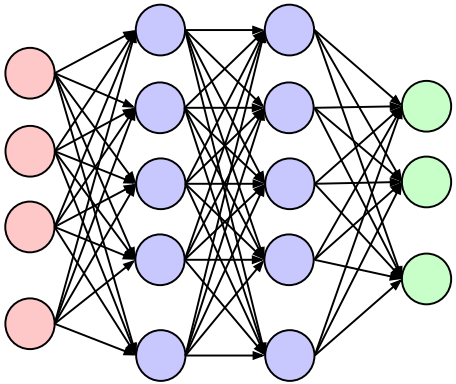


DeepLearning.AI

Pruning

Preparing Models for Deployment in PyTorch

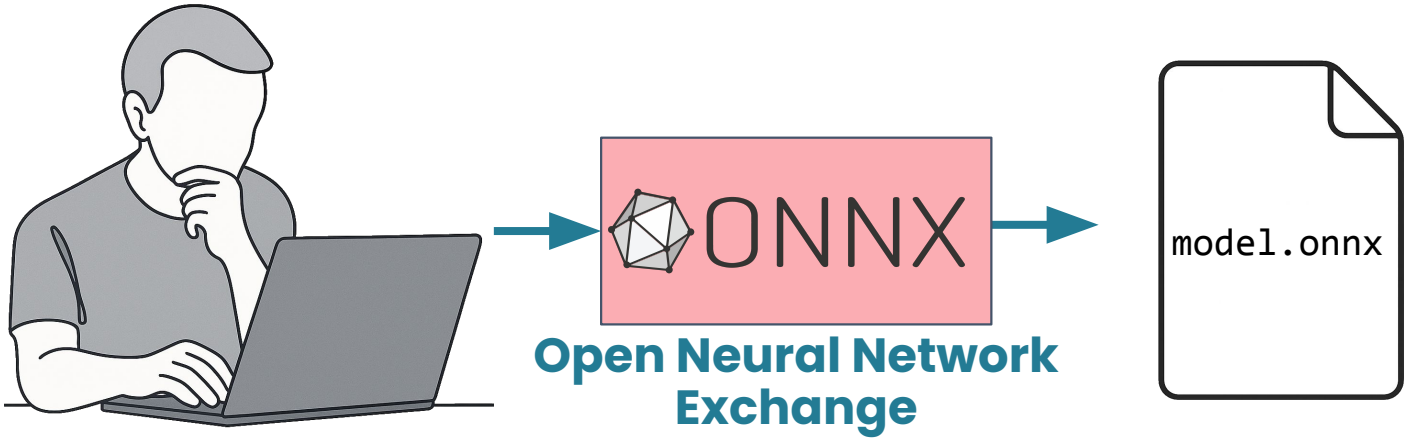
Real-World Models



92% Accuracy

- Fast
- Lightweight
- Efficient

ONNX

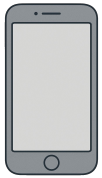


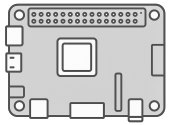
Model Compression

Fast

Lightweight

Efficient





Compression Techniques

- **Pruning**
- **Quantization**

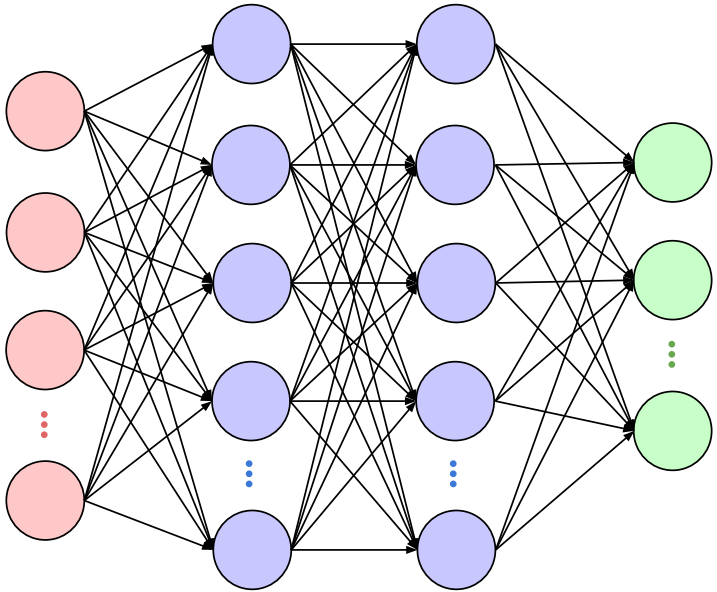
Compression Techniques

- **Pruning**
- **Quantization**

Compression Techniques

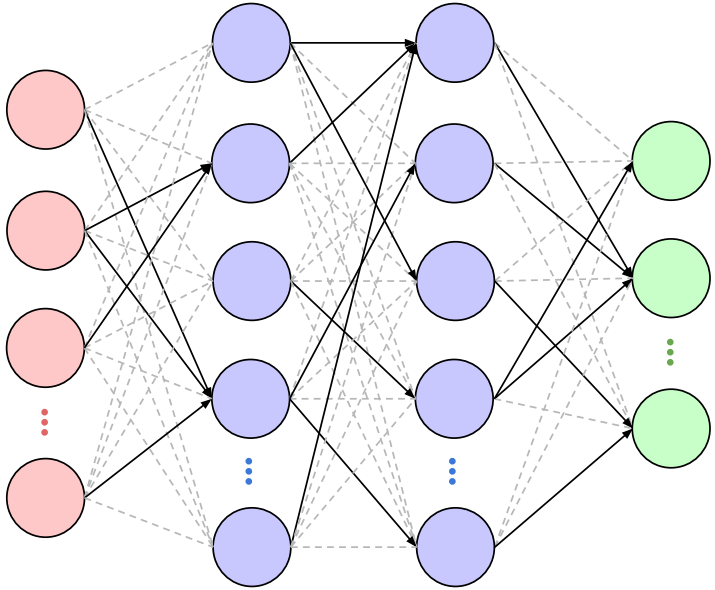
- **Pruning:** Not every part of a model is important
- **Quantization**

Compression Techniques



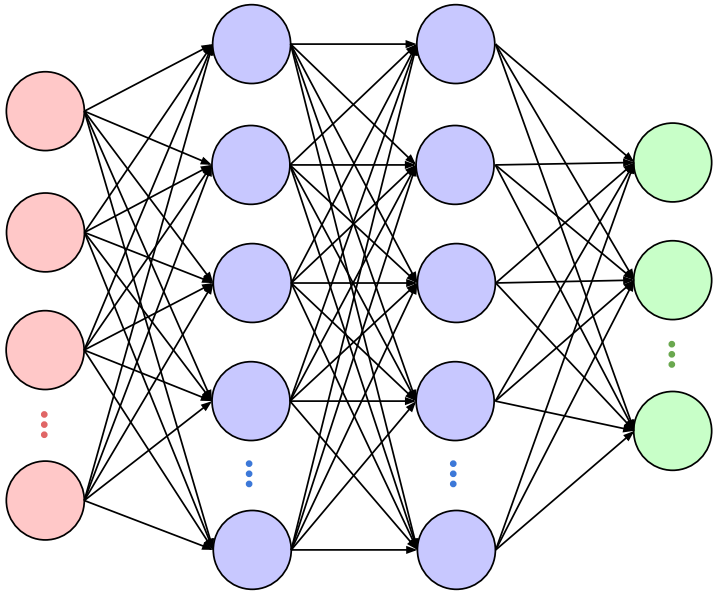
Millions of Parameters

Compression Techniques



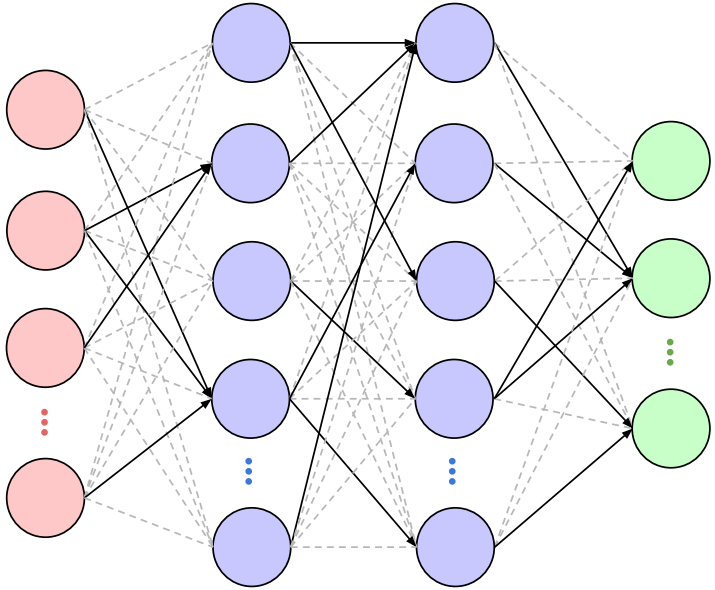
Millions of Parameters

Unstructured Pruning



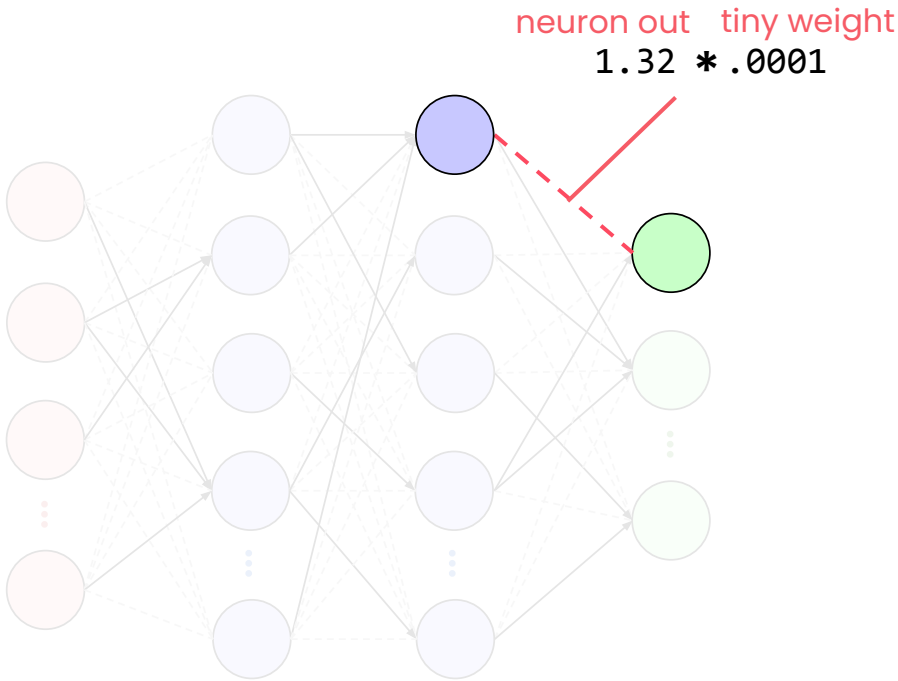
More Parameters than Needed

Unstructured Pruning



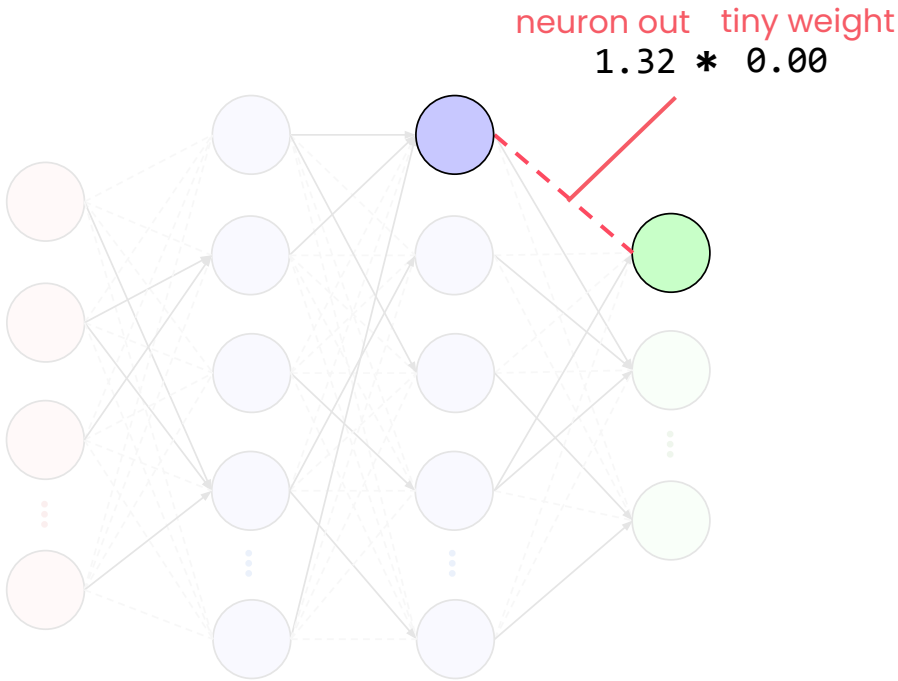
More Parameters than Needed

Unstructured Pruning



More Parameters than Needed

Unstructured Pruning



More Parameters than Needed

Unstructured Pruning

```
model.parameters()
```

Unstructured Pruning

```
import torch.nn.utils.prune as prune

prune.l1_unstructured(model.fc1, name="weight", amount=0.3)
```

Unstructured Pruning

```
import torch.nn.utils.prune as prune

prune.l1_unstructured(model.fc1, name="weight", amount=0.3)
```

Unstructured Pruning

```
import torch.nn.utils.prune as prune

prune.l1_unstructured(model.fc1, name="weight", amount=0.3)
```

Unstructured Pruning

```
import torch.nn.utils.prune as prune

prune.l1_unstructured(model.fc1, name="weight", amount=0.3)

prune.random_unstructured(model.fc1, name="weight", amount=0.3)
```

Unstructured Pruning

```
import torch.nn.utils.prune as prune

prune.l1_unstructured(model.fc1, name="weight", amount=0.3)

prune.random_unstructured(model.fc1, name="weight", amount=0.3)
```

Unstructured Pruning

```
import torch.nn.utils.prune as prune

prune.l1_unstructured(model.fc1, name="bias", amount=0.3)

prune.random_unstructured(model.fc1, name="weight", amount=0.3)
```

Unstructured Pruning

```
import torch.nn.utils.prune as prune

prune.l1_unstructured(model.fc1, name="weight", amount=0.3)

prune.random_unstructured(model.fc1, name="weight", amount=0.3)
```

Unstructured Pruning

```
import torch.nn.utils.prune as prune

prune.l1_unstructured(model.fc1, name="weight", amount=100)

prune.random_unstructured(model.fc1, name="weight", amount=0.3)
```

Unstructured Pruning

```
print("fc1 weights:\n", model.fc1.weight)

tensor([[-0.2658,  0.0693,  0.0689,  0.3122],
        [ 0.4445,  0.1917,  0.1849,  0.1577],
        [ 0.0758,  0.3863,  0.1648,  0.3379],
        . . .]
```

Unstructured Pruning

```
print("fc1 weights:\n", model.fc1.weight)

tensor([[-0.2658,  0.0,     0.0,     0.3122],
        [ 0.4445,  0.1917,  0.1849,  0.0   ],
        [ 0.0,     0.3863,  0.0,     0.3379],
        . . .]
```

same size

same speed

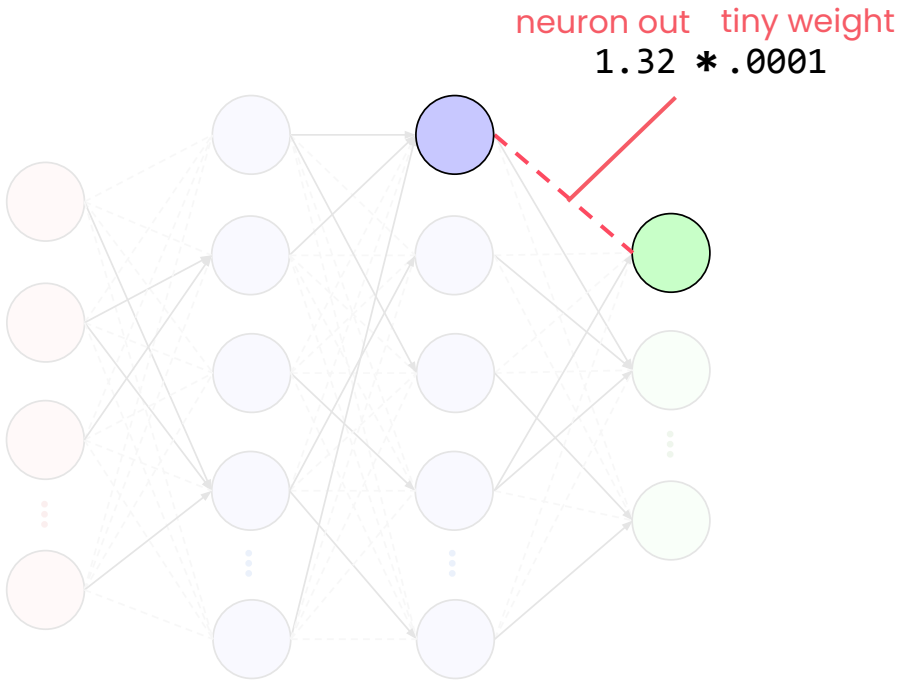
Unstructured Pruning

```
import torch.nn.utils.prune as prune

prune.l1_unstructured(model.fc1, name="weight", amount=100)

prune.random_unstructured(model.fc1, name="weight", amount=0.3)

model.fc1.weight = model.fc1.weight.to_sparse()
```

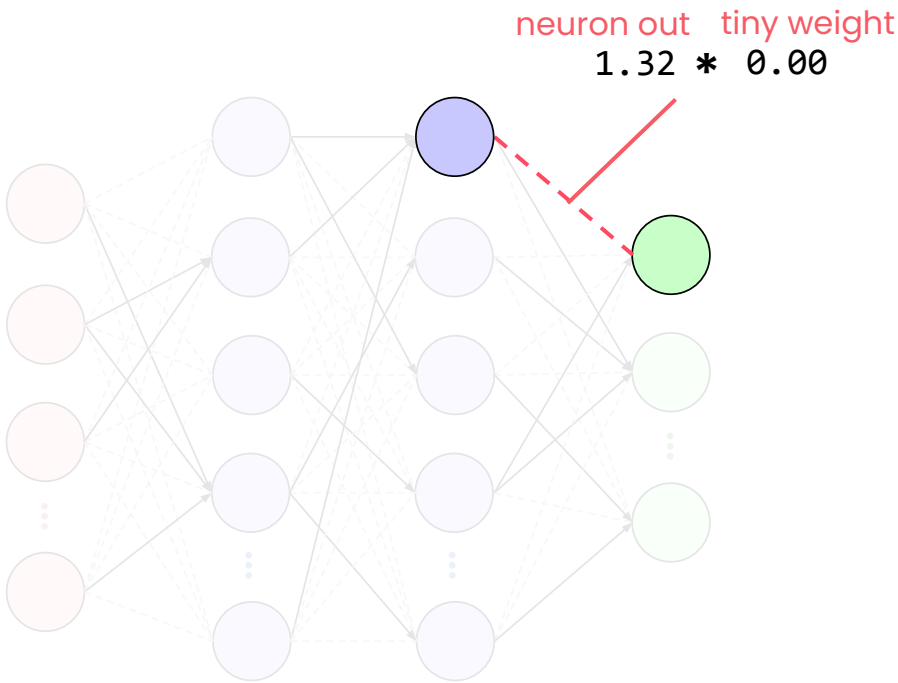
Store non-zero only

Smaller but not optimized

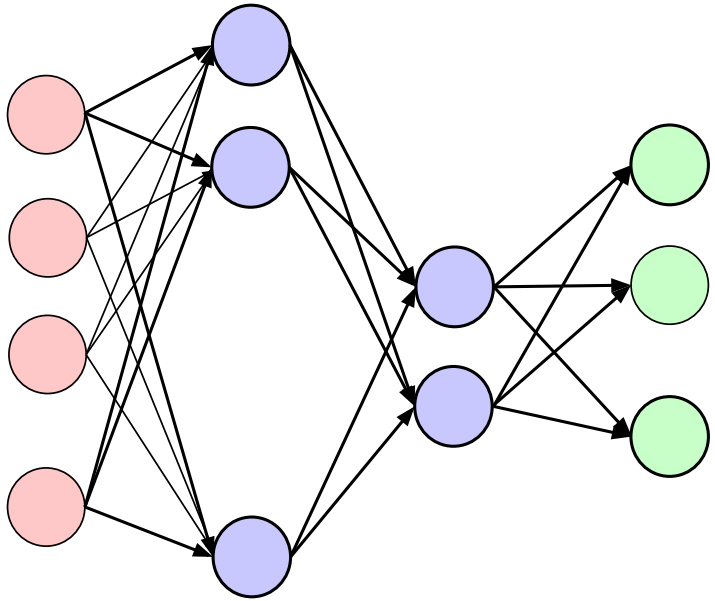
Structured Pruning



Structured Pruning



Structured Pruning



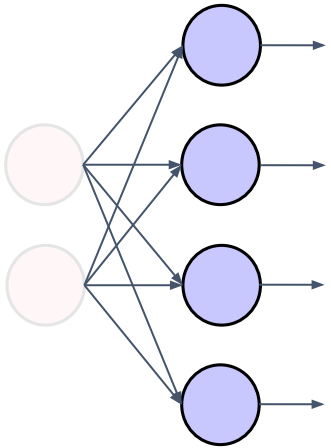
Structured Pruning

```
import torch.nn.utils.prune as prune

prune.ln_structured(model.fc1, name="weight", amount=0.5, n=2, dim=0)
```

Structured Pruning

```
import torch.nn.utils.prune as prune

prune.ln_structured(model.fc1, name="weight", amount=0.5, n=2, dim=0)
```

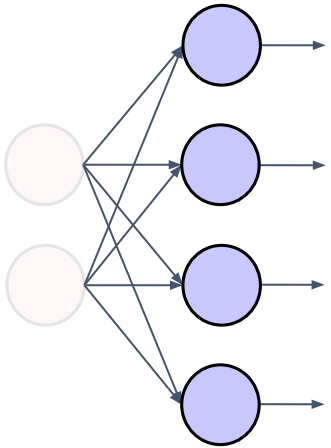
```
nn.Linear(2, 4)
```



Structured Pruning

```
import torch.nn.utils.prune as prune

prune.ln_structured(model.fc1, name="weight", amount=0.5, n=2, dim=0)
```

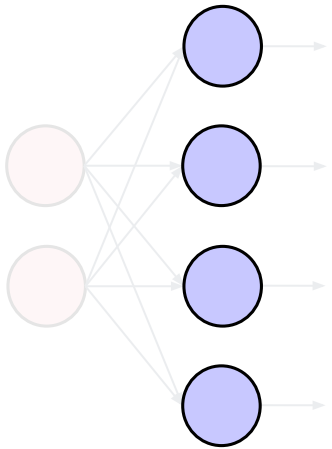
```
nn.Linear(2, 4)
```



Structured Pruning

```
import torch.nn.utils.prune as prune

prune.ln_structured(model.fc1, name="weight", amount=0.5, n=2, dim=0)
```

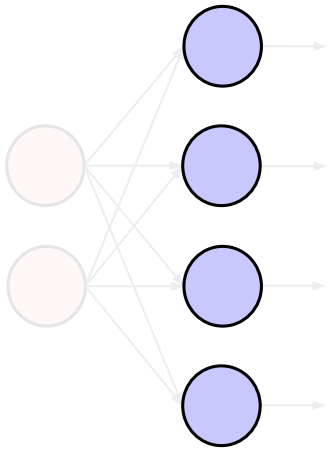
```
nn.Linear(2, 4)
```



Structured Pruning

```
import torch.nn.utils.prune as prune

prune.ln_structured(model.fc1, name="weight", amount=0.5, n=2, dim=0)
```

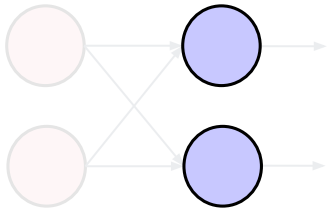
```
nn.Linear(2, 4)
```



Structured Pruning

```
import torch.nn.utils.prune as prune

prune.ln_structured(model.fc1, name="weight", amount=0.5, n=2, dim=0)
```

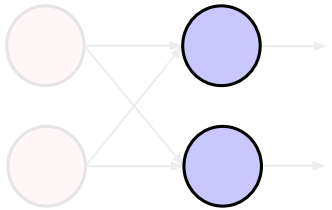
```
nn.Linear(2, 2)
```



Structured Pruning

```
import torch.nn.utils.prune as prune

prune.ln_structured(model.fc1, name="weight", amount=0.5, n=2, dim=1)
```

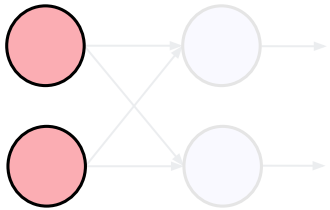
```
nn.Linear(2, 2)
```



Structured Pruning

```
import torch.nn.utils.prune as prune

prune.ln_structured(model.fc1, name="weight", amount=0.5, n=2, dim=1)
```

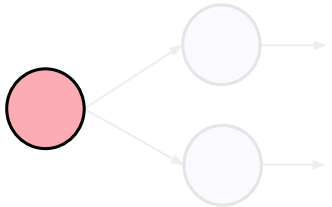
```
nn.Linear(2, 2)
```



Structured Pruning

```
import torch.nn.utils.prune as prune

prune.ln_structured(model.fc1, name="weight", amount=0.5, n=2, dim=1)
```

```
nn.Linear(1, 2)
```



Structured Pruning

```
import torch.nn.utils.prune as prune

prune.ln_structured(model.fc1, name="weight", amount=0.5, n=2, dim=1)


prune.l1_unstructured(model.fc1, name="weight", amount=0.3)
```

```
nn.Linear(1, 2)
```

Structured Pruning

```
import torch.nn.utils.prune as prune

prune.ln_structured(model.fc1, name="weight", amount=0.5, n=2, dim=1)


prune.l1_unstructured(model.fc1, name="weight", amount=0.3)
```

```
nn.Linear(1, 2)
```

Structured Pruning

```
import torch.nn.utils.prune as prune

prune.ln_structured(model.fc1, name="weight", amount=0.5, n=2, dim=0)


prune.l1_unstructured(model.fc1, name="weight", amount=0.3)
```

```
nn.Linear(1, 2)
```

Structured Pruning

```
import torch.nn.utils.prune as prune

prune.ln_structured(model.fc1, name="weight", amount=0.5, n=2, dim=0)


prune.l1_unstructured(model.fc1, name="weight", amount=0.3)
```

```
nn.Linear(1, 2)
```

L1 vs. L2 Structured Pruning

weights:

Neuron A	Neuron B
0.82	0.45
-0.15	0.38
0.08	-0.32
-0.04	0.26

L1 Structured Pruning

```
torch.sum(torch.abs(weights))
```

A: 1.09 < B: 1.41

L2 Structured Pruning

```
torch.sqrt(torch.sum(weights ** 2))
```

A: 0.84 > B: 0.72

Structured Pruning

```
import torch.nn.utils.prune as prune

prune.ln_structured(model.conv1, name="weight", amount=0.5, n=2, dim=0)
```

Structured Pruning

```
import torch.nn.utils.prune as prune

prune.ln_structured(model.conv1, name="weight", amount=0.5, n=2, dim=0)
```

Structured Pruning

```
import torch.nn.utils.prune as prune

prune.ln_structured(model.conv1, name="weight", amount=0.5, n=2, dim=0)
```

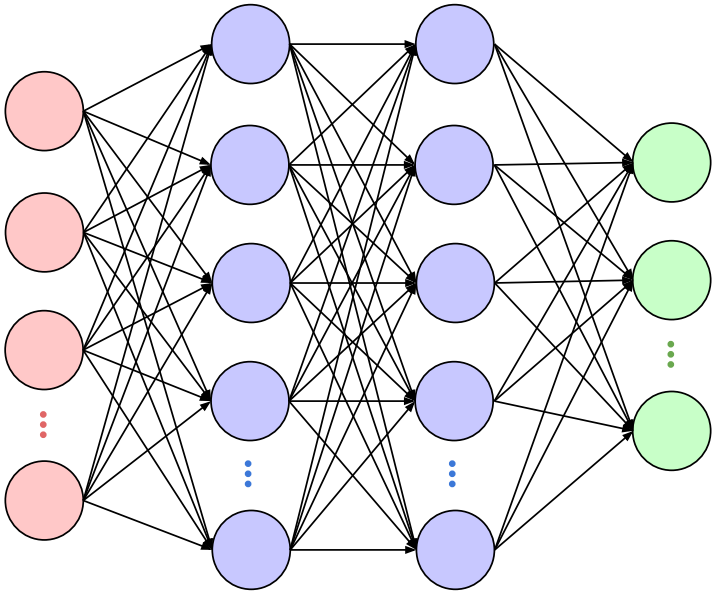
Cut filters

Structured Pruning

```
import torch.nn.utils.prune as prune

prune.ln_structured(model.conv1, name="weight", amount=0.5, n=2, dim=1)
```

Cut inputs to filters

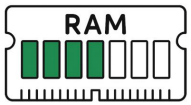
Structured Pruning



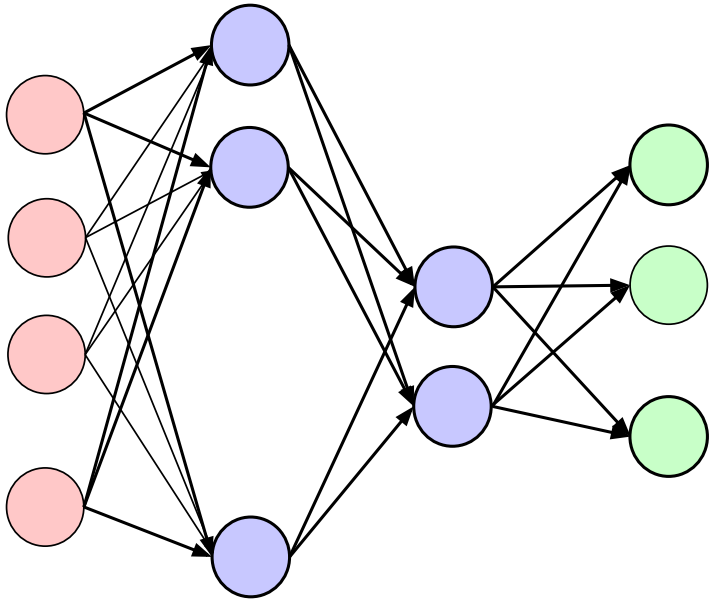
Structured Pruning

Fewer parameters

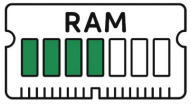
Fewer feature maps



Larger impact on accuracy

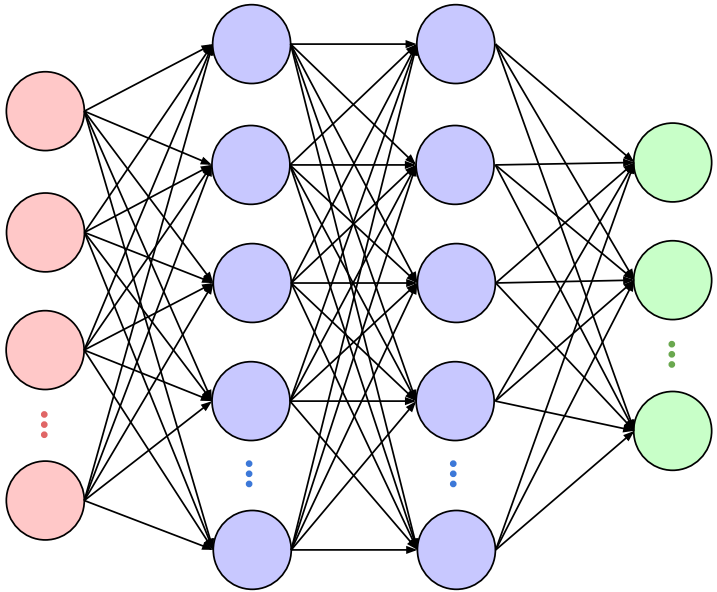


Pruning

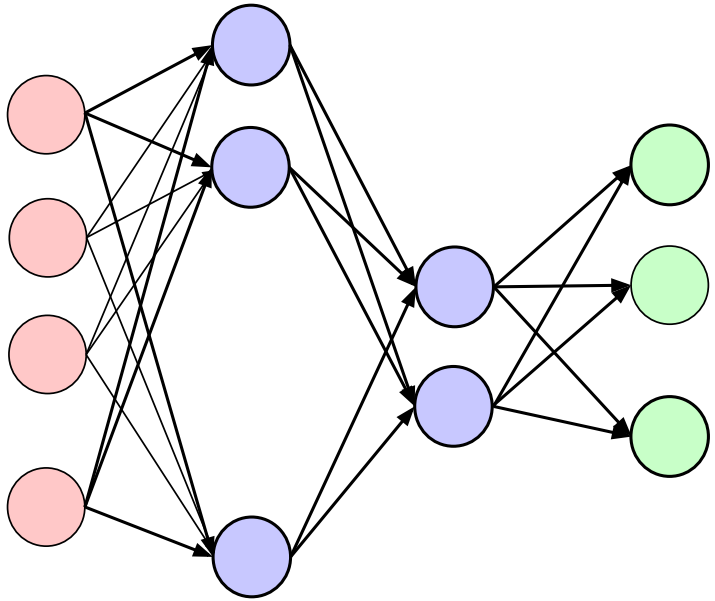


Metric	Before pruning	After pruning
Accuracy	**95 %**	**93 %**
Inference time	**100 ms**	**70 ms**
File size	**50 MB**	**25 MB**

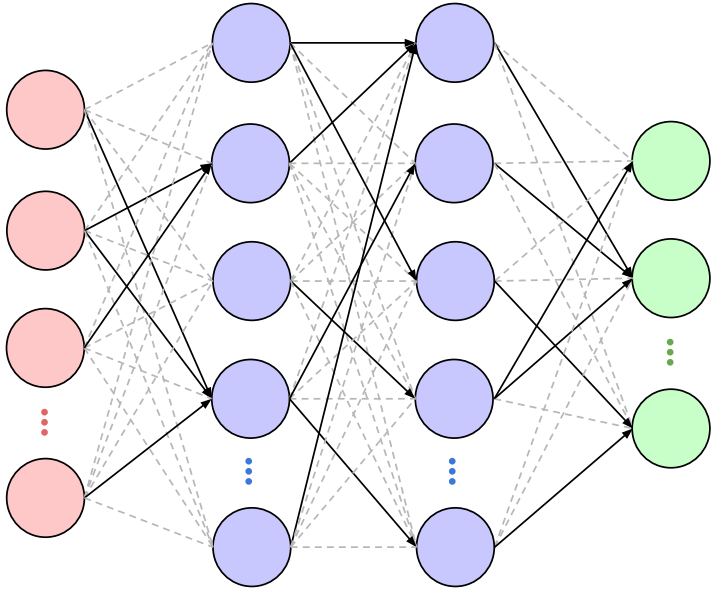
Pruning



Pruning



Pruning

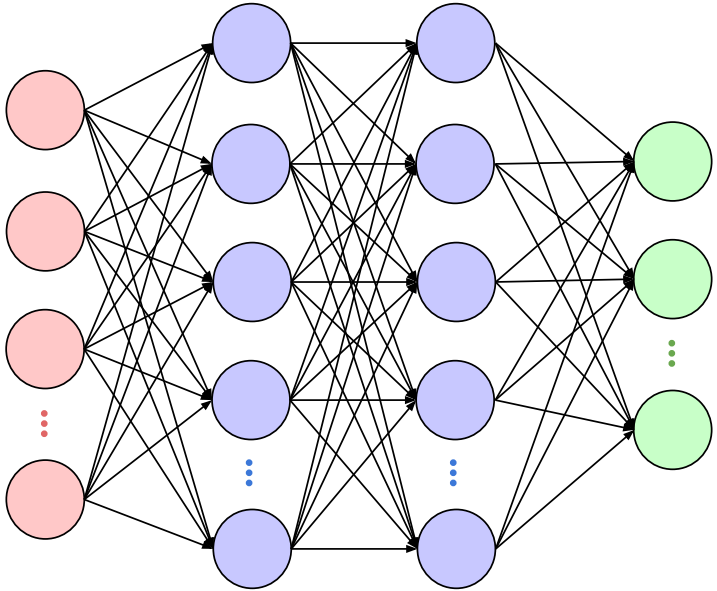


DeepLearning.AI

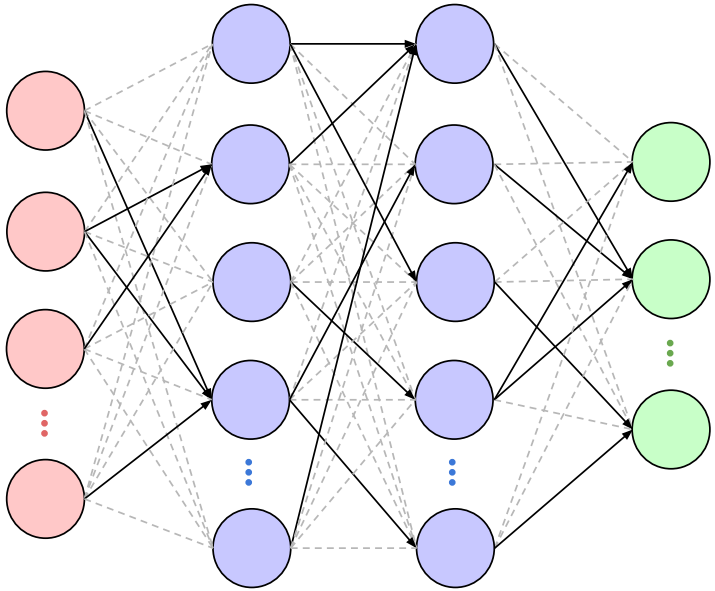
Static and Dynamic Quantization

Preparing Models for Deployment in PyTorch

Pruning



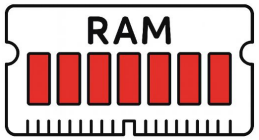
Pruning



32-bit Weights

Model Weight

```
0.0283746
```



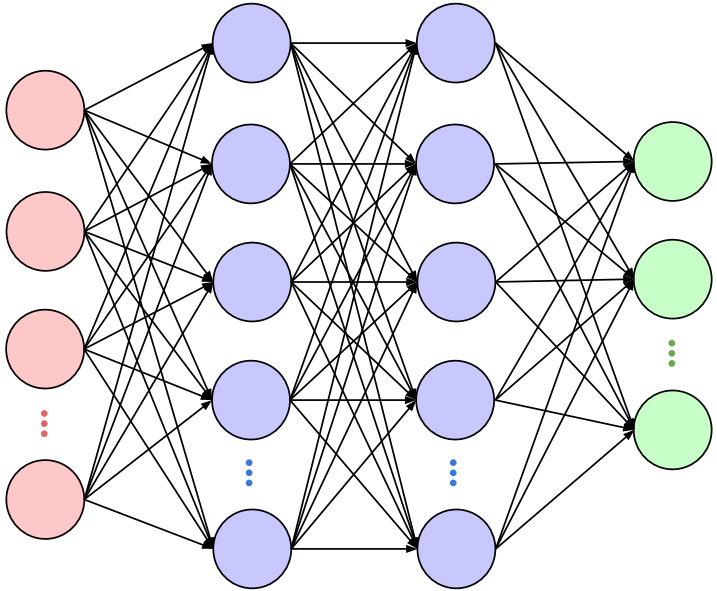
32-bit floating point value

0	0	1	1	1	1	0	0	1	1	1	0	1	0	0	0	0	1	1	1	0	0	0	0	1	0	1	1	1	1	1	0

During Training

Model Weight

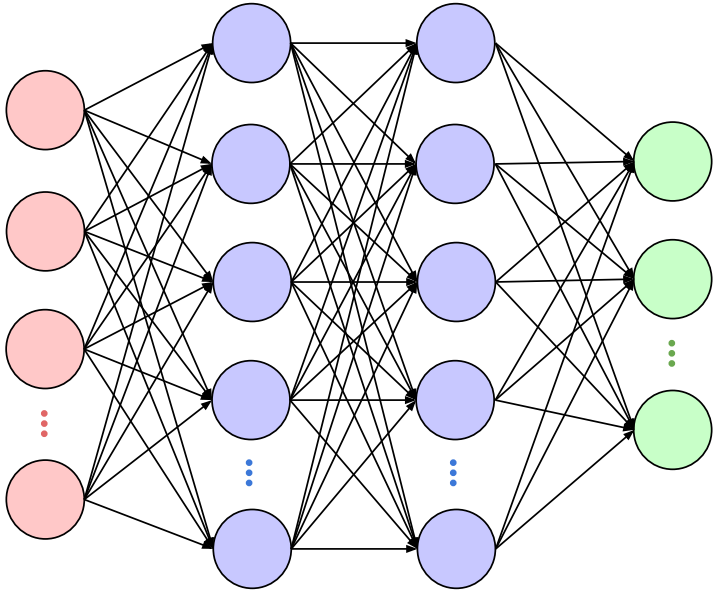
```
0.0283746
```



During Training

Model Weight

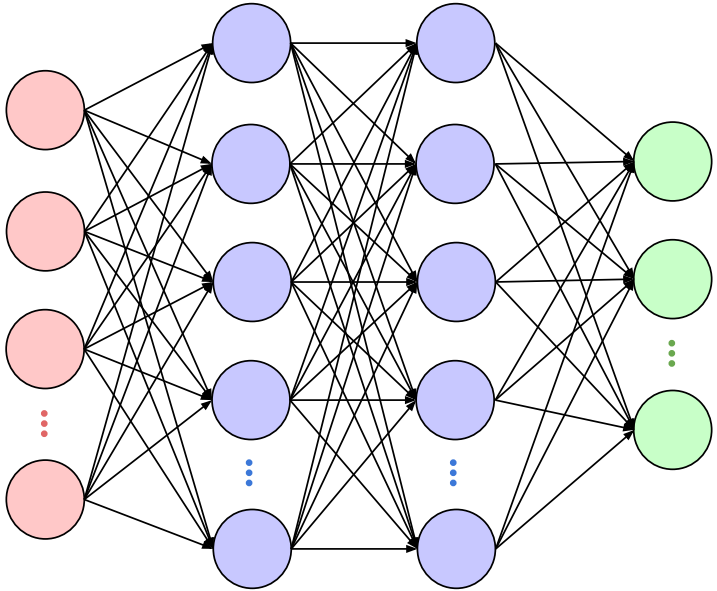
`0.0283744`



During Training

Model Weight

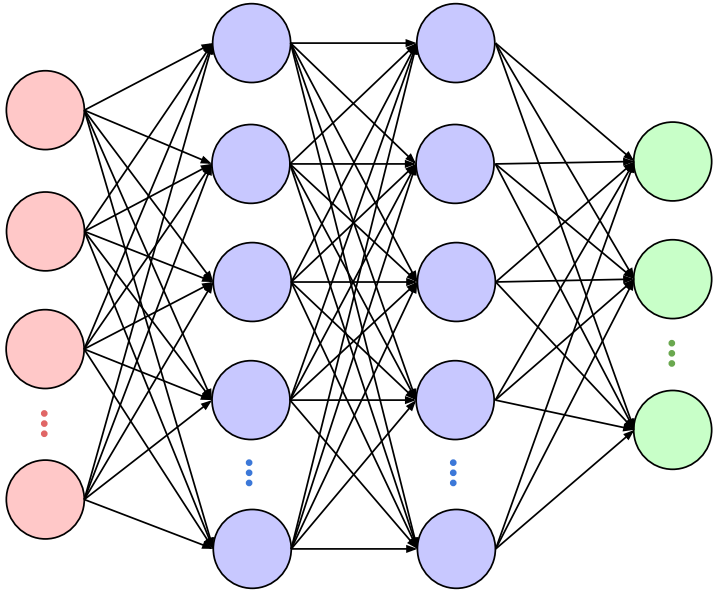
`0.0283732`



During Training

Model Weight

`0.0283668`

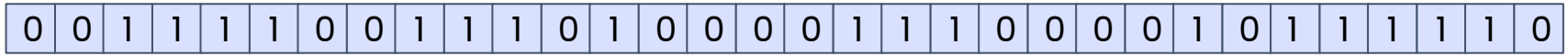


During Training

Model Weight

```
0.0283668
```

32-bit floating point value



Training Complete

Model Weight

`0.0283668`

32-bit floating point value

0	0	1	1	1	1	0	0	1	1	1	0	1	0	0	0	0	1	1	1	0	0	0	0	1	0	1	1	1	1	1	0

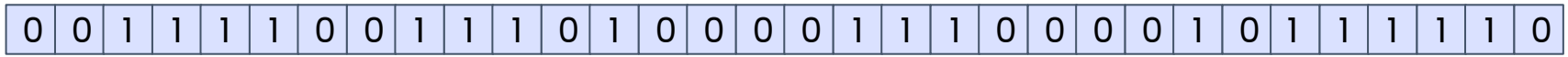
Still need this much precision?

Training Complete

Model Weight

```
0.0283668
```

32-bit floating point value

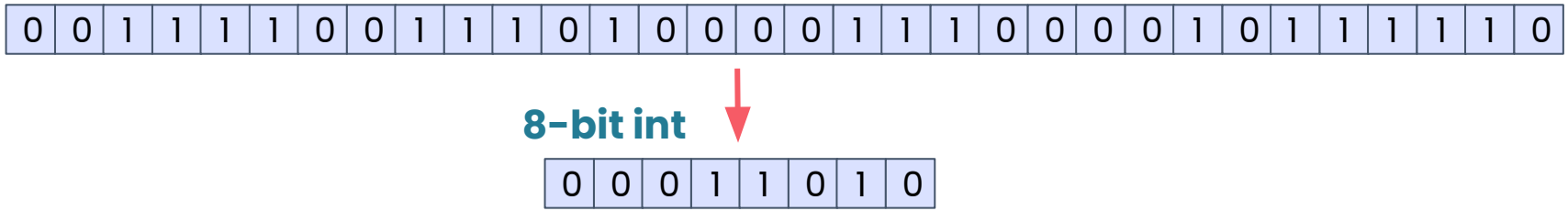


Training Complete

Model Weight

26

32-bit floating point value



Noise Tolerance

Model Weight

```
0.0283668
```

Class Predictions

```
Cat:  0.82
Dog:  0.17
Bird: 0.01
```

Noise Tolerance

Model Weight

```
0.0280000
```

Class Predictions

```
Cat:  0.82
Dog:  0.17
Bird: 0.01
```

Noise Tolerance

Model Weight

```
0.0280000
```

Class Predictions

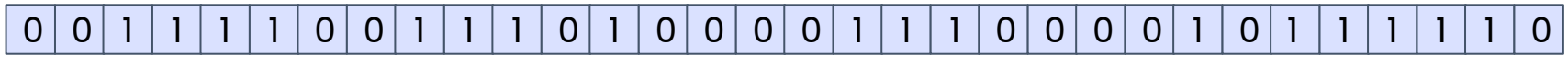
```
Cat:  0.81
Dog:  0.18
Bird: 0.01
```

Quantization

Model Weight

```
0.0283668
```

32-bit float

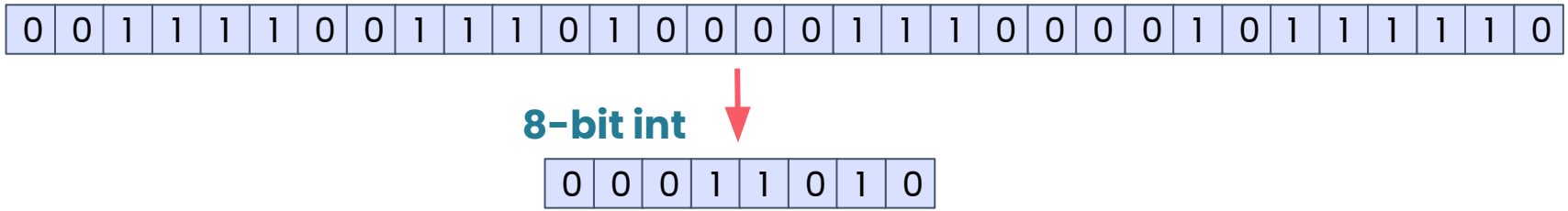


Quantization

Model Weight

```
26
```

32-bit float



Quantization

```
class SimpleModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Linear(4, 1)

    def forward(self, x):
        x = self.layer1(x)
        return x
```

Quantization

```
class SimpleModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Linear(4, 1)

    def forward(self, x):
        x = self.layer1(x)
        return x
```

Quantizing Weights

```
weights = model.layer1.weight.data
print(weights)
> tensor([0.12537, -0.3749, 0.7526, -1.0412])
```

Quantizing Weights

```
weights = model.layer1.weight.data
print(weights)
> tensor([0.12537, -0.3749, 0.7526, -1.0412])
```

Quantizing Weights

```
weights = model.layer1.weight.data
print(weights)
> tensor([0.12537, -0.3749, 0.7526, -1.0412])


bias = model.layer1.bias.data
```

bias not quantized

Quantizing Weights

```
weights = model.layer1.weight.data
print(weights)
> tensor([0.12537, -0.3749, 0.7526, -1.0412])
```

Quantizing Weights

```
weights = model.layer1.weight.data
print(weights)
> tensor([0.12537, -0.3749, 0.7526, -1.0412])
```

int8 range: -128 to 127

Quantizing Weights

```
weights = model.layer1.weight.data
print(weights)
> tensor([0.12537, -0.3749, 0.7526, -1.0412])


scale = weights.abs().max()/127  # abs(-1.0412)/127
```

int8 range: -128 to 127

Quantizing Weights

```
weights = model.layer1.weight.data
print(weights)
> tensor([0.12537, -0.3749, 0.7526, -1.0412])


scale = weights.abs().max()/127  # abs(-1.0412)/127
print(scale)
> tensor(0.0082)
```

int8 range: -128 to 127

Quantizing Weights

```
weights = model.layer1.weight.data
print(weights)
> tensor([0.12537, -0.3749, 0.7526, -1.0412])


scale = weights.abs().max()/127  # abs(-1.0412)/127
print(scale)
> tensor(0.0082)

int8_weights = torch.round(weights/scale).to(torch.int8)
```

int8 range: -128 to 127

Quantizing Weights

```
weights = model.layer1.weight.data
print(weights)
> tensor([0.12537, -0.3749, 0.7526, -1.0412])


scale = weights.abs().max()/127  # abs(-1.0412)/127
print(scale)
> tensor(0.0082)

int8_weights = torch.round(weights/scale).to(torch.int8)
print(int8_weights)
> tensor([15, -46, 92, -127], dtype=torch.int8)
```

int8 range: -128 to 127

Quantizing Weights

```
weights = model.layer1.weight.data
print(weights)
> tensor([0.12537, -0.3749, 0.7526, -1.0412])


scale = weights.abs().max()/127  # abs(-1.0412)/127
print(scale)
> tensor(0.0082)

int8_weights = torch.round(weights/scale).to(torch.int8)
print(int8_weights)
> tensor([15, -46, 92, -127], dtype=torch.int8)
```

int8 range: -128 to 127

Quantizing Weights

```
weights = model.layer1.weight.data
print(weights)
> tensor([0.12537, -0.3749, 0.7526, -1.0412])


scale = weights.abs().max()/127  # abs(-1.0412)/127
print(scale)
> tensor(0.0082)

int8_weights = torch.round(weights/scale).to(torch.int8)
print(int8_weights)
> tensor([15, -46, 92, -127], dtype=torch.int8)
```

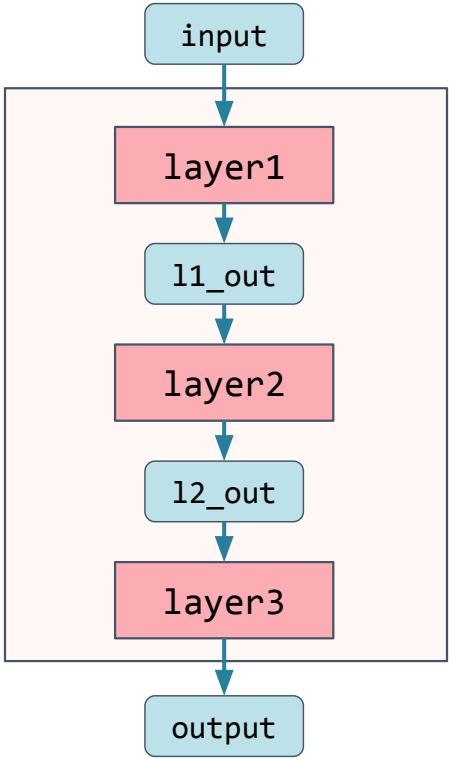
int8 range: -128 to 127

Quantizing Weights

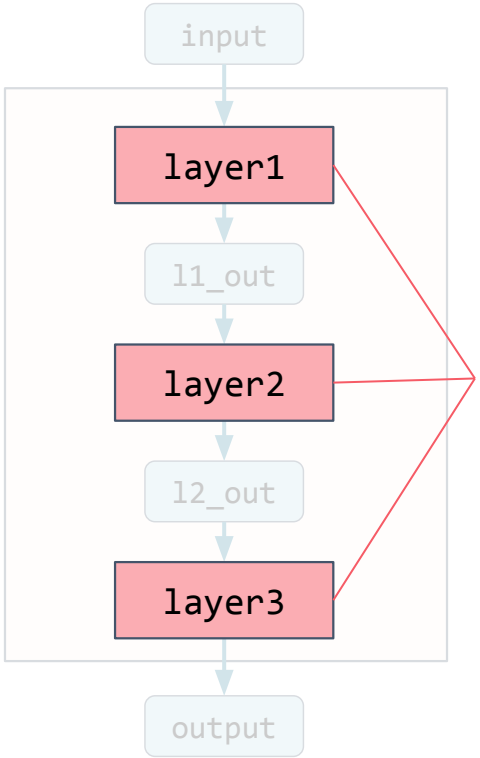
```
weights = model.layer1.weight.data
print(weights)
> tensor([0.12537, -0.3749, 0.7526, -1.0412])


scale = weights.abs().max()/127  # abs(-1.0412)/127
print(scale)
> tensor(0.0082)

int8_weights = torch.round(weights/scale).to(torch.int8)
print(int8_weights)
> tensor([15, -46, 92, -127], dtype=torch.int8)

dequantized = int8_weights.float * scale
```

int8 range: -128 to 127

Quantization



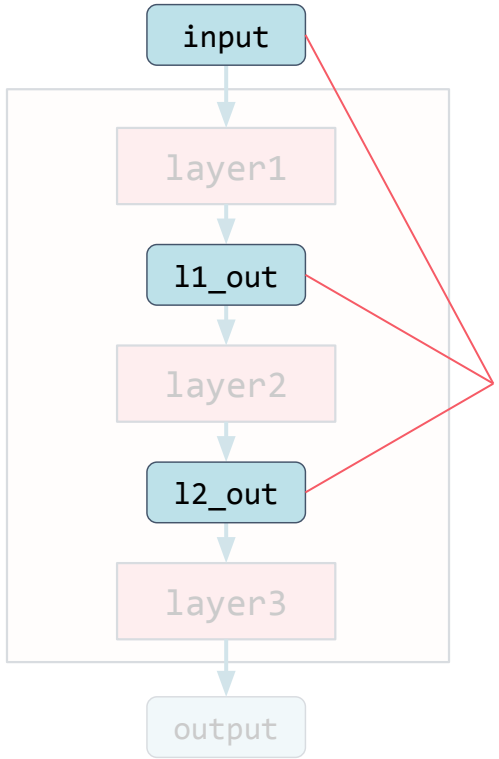
Quantization



Known Weights:

Scale values can be precomputed

Quantization

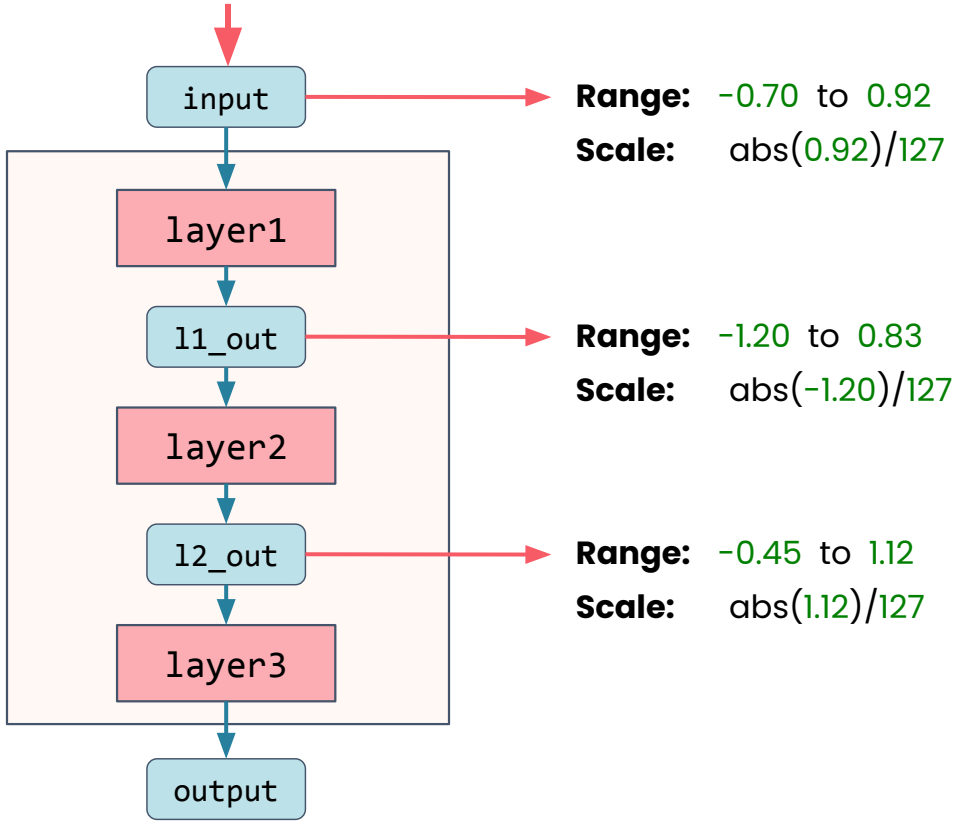


How do you quantize numbers that only appear at runtime?

You can't derive scale values ahead of time

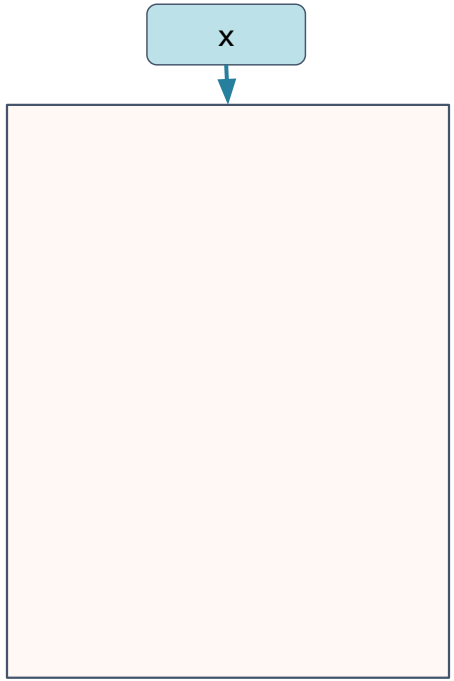
Calibration

```
# Sample data

([[ 0.25, -0.40,  0.80,  0.10],
  [-0.10,  0.30, -0.50,  0.60],
  [ 0.75,  0.20, -0.10, -0.30],
  [-0.45, -0.25,  0.50,  0.40],
  [ 0.10,  0.90,  0.20, -0.60],
  [-0.20,  0.15, -0.70,  0.50],
              . . .
```



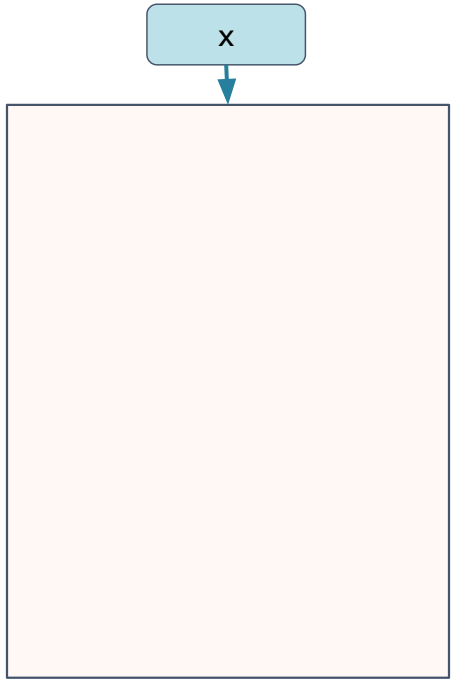
Static Quantization

```
class QuantizedCNN(nn.Module):
    def __init__(self):
        super(QuantizedCNN, self).__init__()




    def forward(self, x):



        return x
```



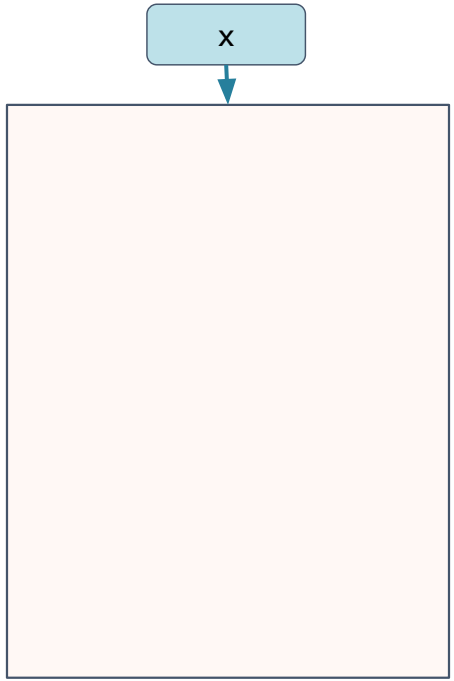
Static Quantization

```
class QuantizedCNN(nn.Module):
    def __init__(self):
        super(QuantizedCNN, self).__init__()

        self.quant = torch.quantization.QuantStub()



    def forward(self, x):



        return x
```



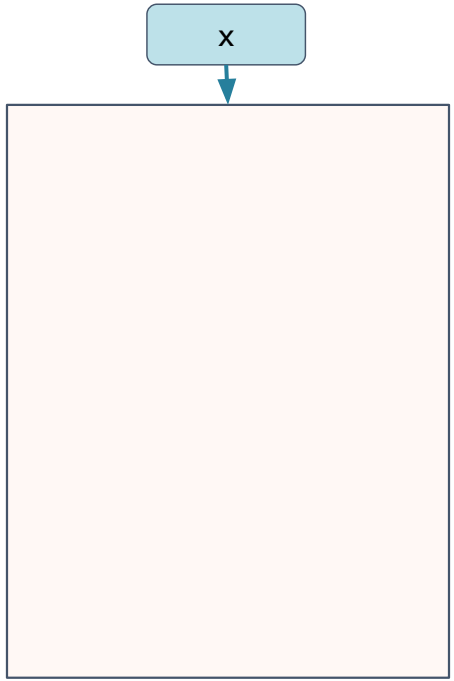
Static Quantization

```
class QuantizedCNN(nn.Module):
    def __init__(self):
        super(QuantizedCNN, self).__init__()

        self.quant = torch.quantization.QuantStub()
        . . .   # Model Layers
        self.dequant = torch.quantization.DeQuantStub()


    def forward(self, x):



        return x
```



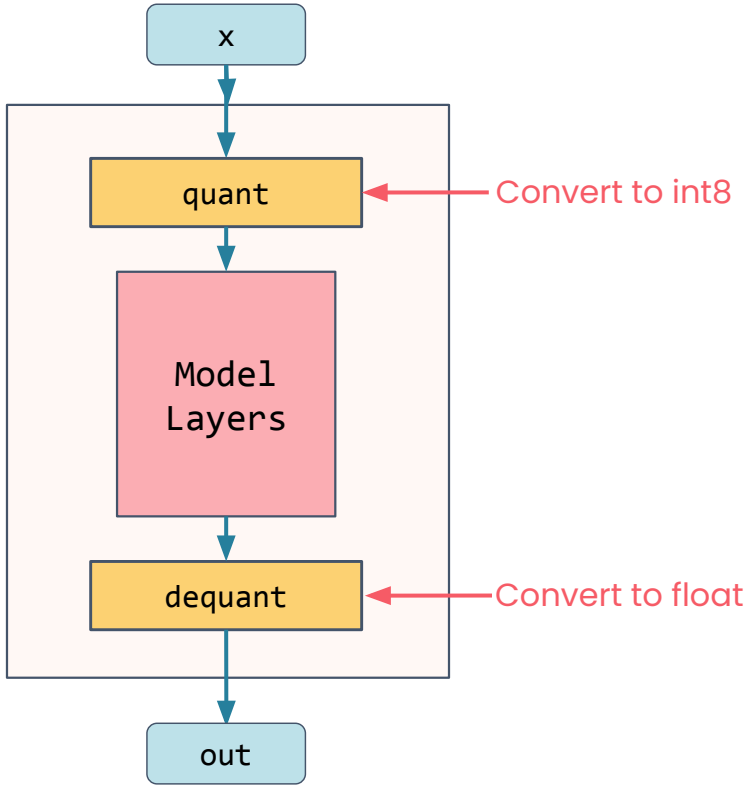
Static Quantization

```
class QuantizedCNN(nn.Module):
    def __init__(self):
        super(QuantizedCNN, self).__init__()

        self.quant = torch.quantization.QuantStub()
        . . .   # Model Layers
        self.dequant = torch.quantization.DeQuantStub()


    def forward(self, x):



        return x
```



Static Quantization

```
class QuantizedCNN(nn.Module):
    def __init__(self):
        super(QuantizedCNN, self).__init__()

        self.quant = torch.quantization.QuantStub()
        . . .   # Model Layers
        self.dequant = torch.quantization.DeQuantStub()


    def forward(self, x):
        x = self.quant(x)
          . . .   # Model Layers
        x = self.dequant(x)
        return x
```



Preparing the Model

```
quantized_static_model = QuantizedCNN()

quantized_static_model.load_state_dict(baseline_model.state_dict())
quantized_static_model.eval()
```

Preparing the Model

```
quantized_static_model = QuantizedCNN()

quantized_static_model.load_state_dict(baseline_model.state_dict())
quantized_static_model.eval()
```

Preparing the Model

```
quantized_static_model = QuantizedCNN()

quantized_static_model.load_state_dict(baseline_model.state_dict())
quantized_static_model.eval()
```

Preparing the Model

```
quantized_static_model = QuantizedCNN()

quantized_static_model.load_state_dict(baseline_model.state_dict())
quantized_static_model.eval()


# Set as ‘x86’ in the lab - see lab for details
quantized_static_model.qconfig = torch.quantization.get_default_qconfig('fbgemm')
```

Preparing the Model

```
quantized_static_model = QuantizedCNN()

quantized_static_model.load_state_dict(baseline_model.state_dict())
quantized_static_model.eval()


# Set as ‘x86’ in the lab - see lab for details
quantized_static_model.qconfig = torch.quantization.get_default_qconfig('fbgemm')
```

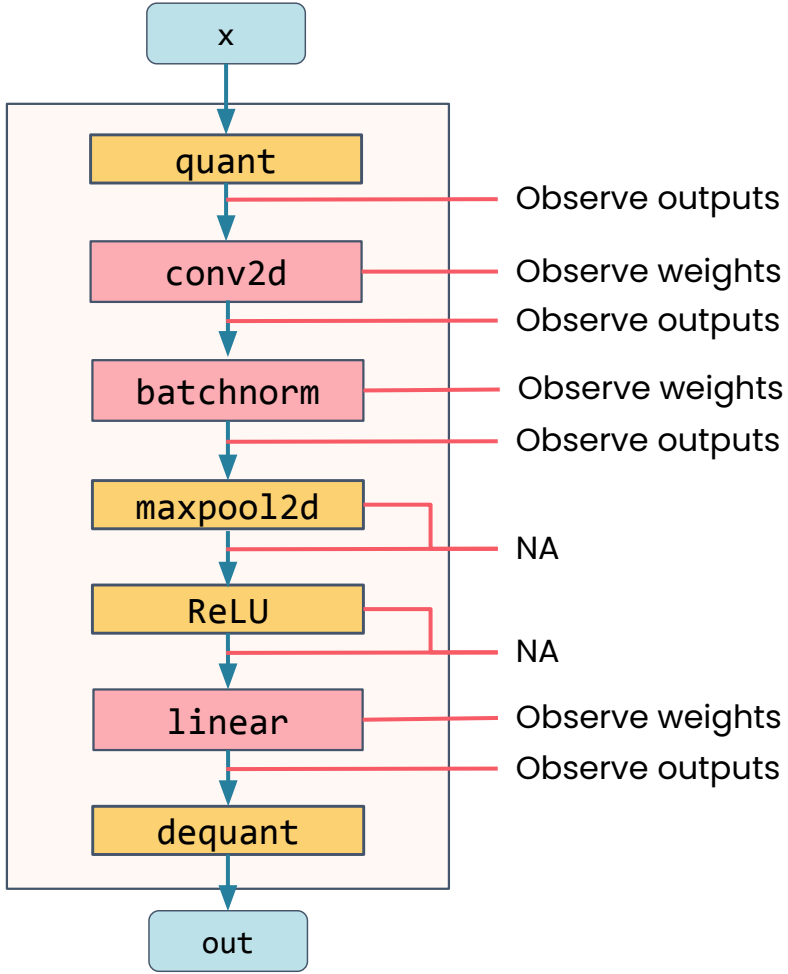
Preparing the Model

```
quantized_static_model = QuantizedCNN()

quantized_static_model.load_state_dict(baseline_model.state_dict())
quantized_static_model.eval()


# Set as ‘x86’ in the lab - see lab for details
quantized_static_model.qconfig = torch.quantization.get_default_qconfig('fbgemm')

torch.quantization.prepare(quantized_static_model, inplace=True)
```

Preparing the Model

```
quantized_static_model = QuantizedCNN()

quantized_static_model.load_state_dict(baseline_model.state_dict())
quantized_static_model.eval()


# Set as ‘x86’ in the lab - see lab for details
quantized_static_model.qconfig = torch.quantization.get_default_qconfig('fbgemm')

torch.quantization.prepare(quantized_static_model, inplace=True)

calibrate(quantized_static_model, testloader)
```

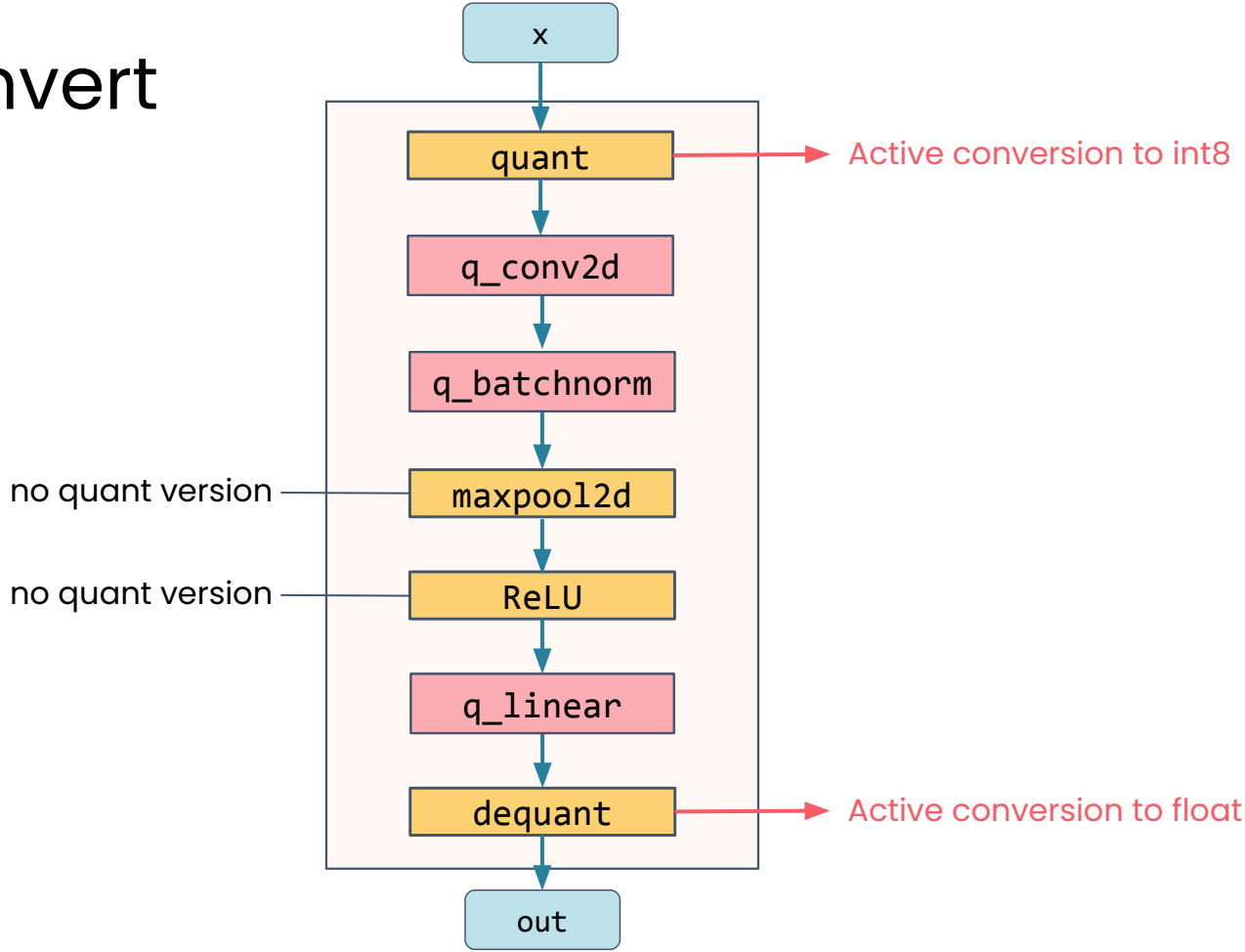
Observers


x
quant
conv2d
batchnorm
maxpool2d
ReLU
linear
dequant
out
Observe outputs
Observe weights
Observe outputs
Observe weights
Observe outputs
NA
NA
Observe weights
Observe outputs

Preparing the Model

```
quantized_static_model = QuantizedCNN()

quantized_static_model.load_state_dict(baseline_model.state_dict())
quantized_static_model.eval()


# Set as ‘x86’ in the lab - see lab for details
quantized_static_model.qconfig = torch.quantization.get_default_qconfig('fbgemm')

torch.quantization.prepare(quantized_static_model, inplace=True)

calibrate(quantized_static_model, testloader)

torch.quantization.convert(quantized_static_model, inplace=True)
```

After Convert



Preparing the Model

```
quantized_static_model = QuantizedCNN()

quantized_static_model.load_state_dict(baseline_model.state_dict())
quantized_static_model.eval()


# Set as ‘x86’ in the lab - see lab for details
quantized_static_model.qconfig = torch.quantization.get_default_qconfig('fbgemm')

torch.quantization.prepare(quantized_static_model, inplace=True)

calibrate(quantized_static_model, testloader)

torch.quantization.convert(quantized_static_model, inplace=True)
```

Dynamic Quantization

```
baseline_model.eval()
```

Dynamic Quantization

```
baseline_model.eval()


quantized_model = torch.quantization.quantize_dynamic(
    baseline_model,
    {nn.Linear},
    dtype=torch.qint8
)
```

Dynamic Quantization

```
baseline_model.eval()


quantized_model = torch.quantization.quantize_dynamic(
    baseline_model,
    {nn.Linear},
    dtype=torch.qint8
)
```

Dynamic Quantization

```
baseline_model.eval()
```

No calibration step

```
quantized_model = torch.quantization.quantize_dynamic(
    baseline_model,
    {nn.Linear},
    dtype=torch.qint8
)
```

Dynamic vs. Static Quantization

- **Dynamic doesn't support all layers** (e.g. conv2d)
- **Static and Dynamic only runs on CPUs**

Dynamic vs. Static Quantization

Dynamic:

- Minimal code
- No calibration
- nn.Linear heavy models

Static:

- More setup
- Bigger speed and memory savings
- Works with convolutions (CNNs)

DeepLearning.AI

Quantization Aware Training

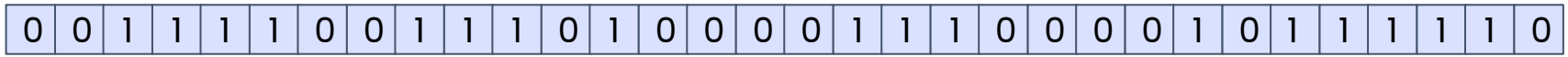
Preparing Models for Deployment in PyTorch

Quantization

Model Weight

```
0.0283668
```

32-bit floating point value

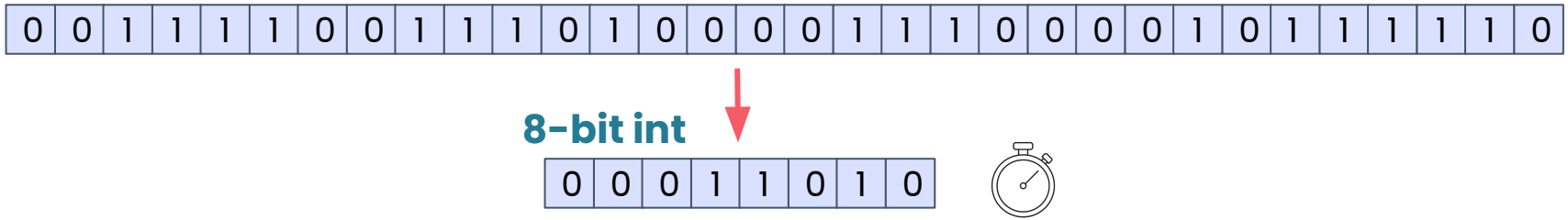


Quantization

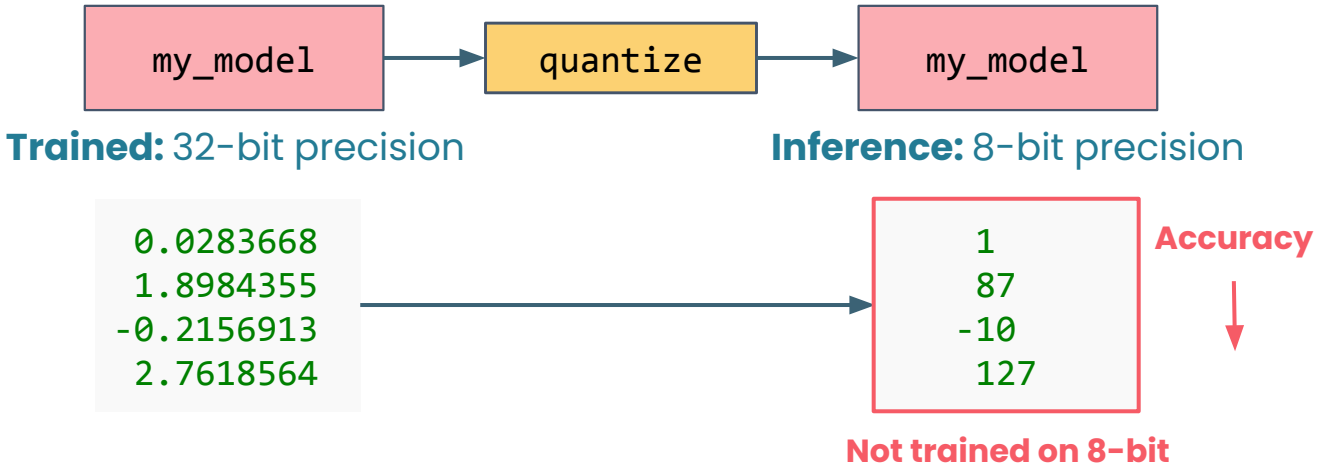
Model Weight

```
26
```

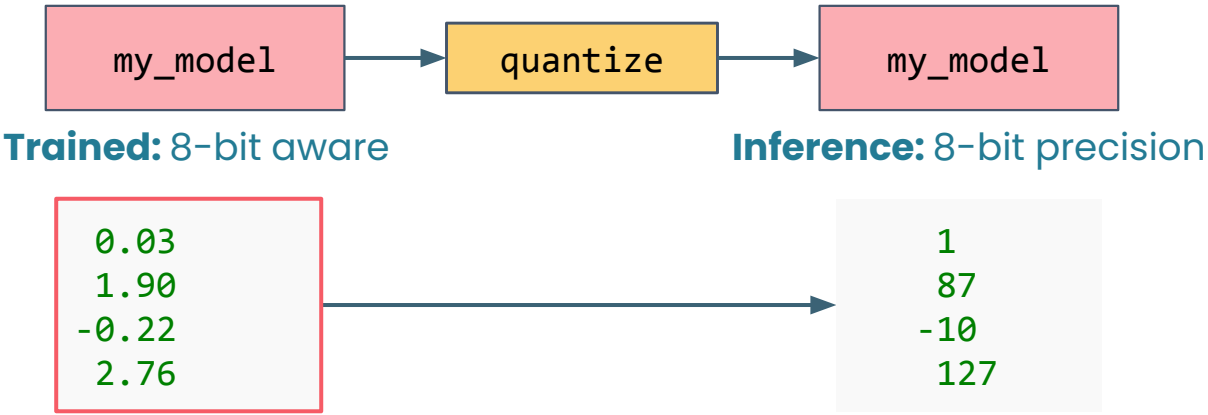
32-bit floating point value



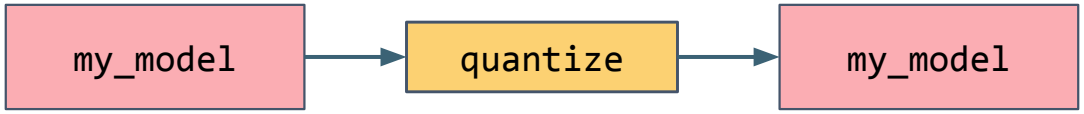
Quantization



Quantization



Quantization Aware Training (QAT)



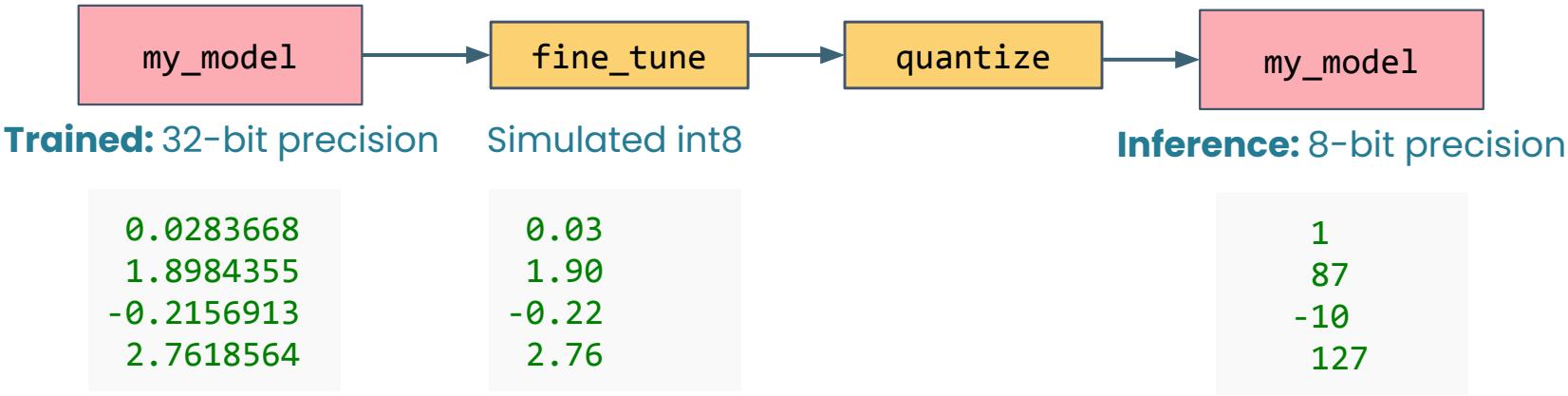
Trained: 32-bit precision

```
 0.0283668
 1.8984355
-0.2156913
 2.7618564
```

Inference: 8-bit precision

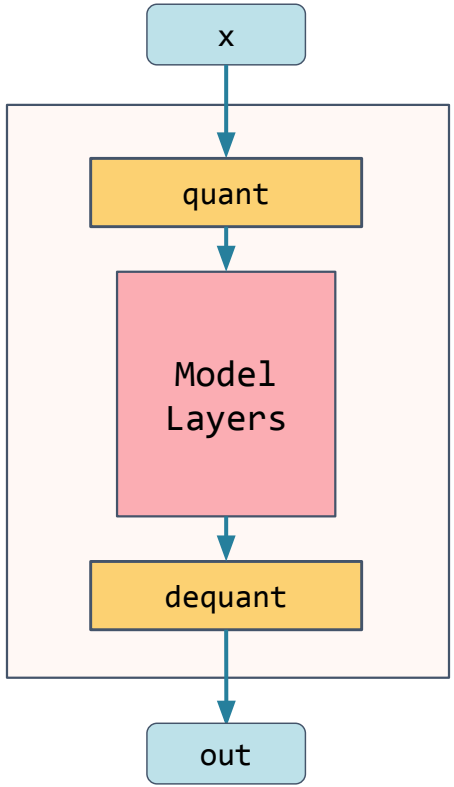
```
  1
 87
-10
 127
```

Quantization Aware Training (QAT)



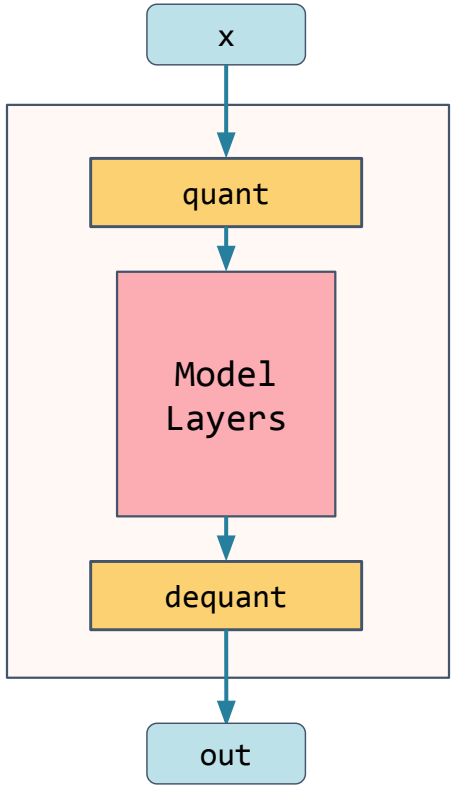
QAT

```
class QATCNN(nn.Module):
    def __init__(self):
        super(QATCNN, self).__init__()

        self.quant = torch.quantization.QuantStub()
        . . .   # Model Layers
        self.dequant = torch.quantization.DeQuantStub()


    def forward(self, x):
        x = self.quant(x)
          . . .   # Model Layers
        x = self.dequant(x)
        return x
```



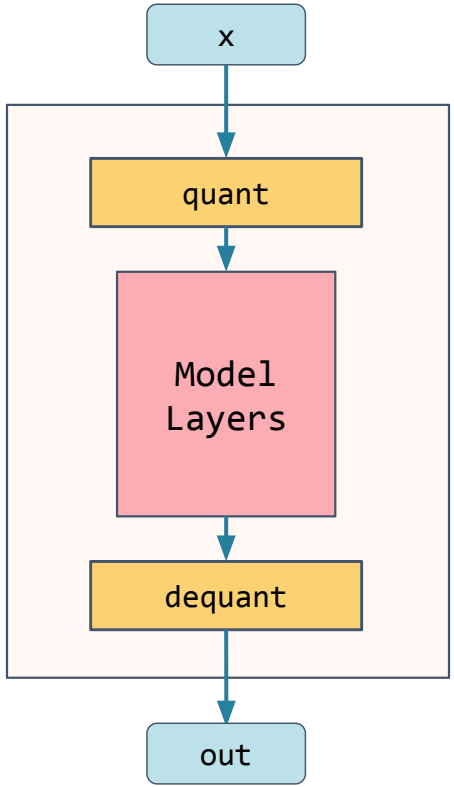
QAT

```
class QATCNN(nn.Module):
    def __init__(self):
        super(QATCNN, self).__init__()

        self.quant = torch.quantization.QuantStub()
        . . .   # Model Layers
        self.dequant = torch.quantization.DeQuantStub()


    def forward(self, x):
        x = self.quant(x)
          . . .   # Model Layers
        x = self.dequant(x)
        return x
```



QAT

```
class QATCNN(nn.Module):
    def __init__(self):
        super(QATCNN, self).__init__()

        self.quant = torch.quantization.QuantStub()
        . . .   # Model Layers
        self.dequant = torch.quantization.DeQuantStub()


    def forward(self, x):
        x = self.quant(x)
          . . .   # Model Layers
        x = self.dequant(x)
        return x
```



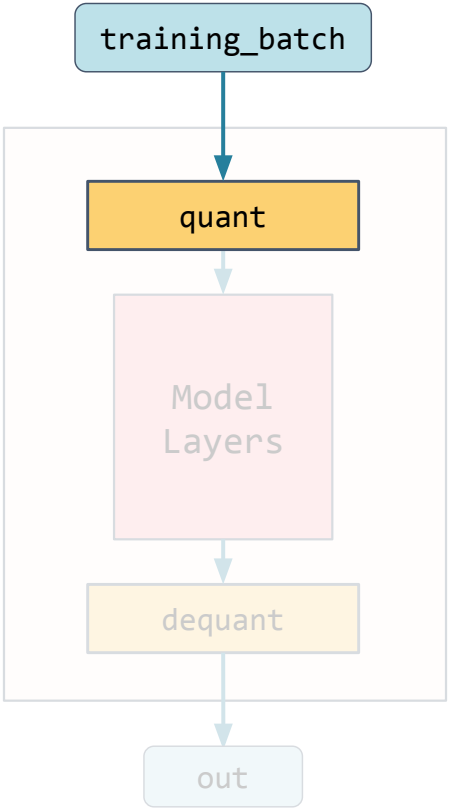
QAT

```
# Prepare the model for Quantization-Aware Training.
qat_model = torch.quantization.prepare_qat(qat_model)
```

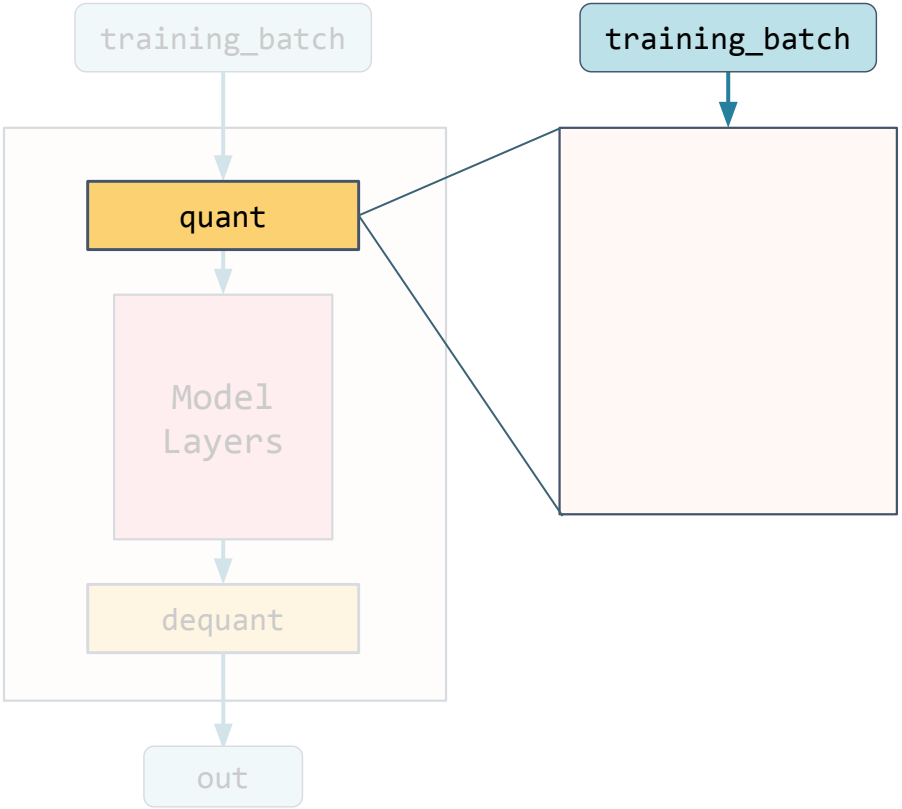
QAT

```
# Prepare the model for Quantization-Aware Training.
qat_model = torch.quantization.prepare_qat(qat_model)
```

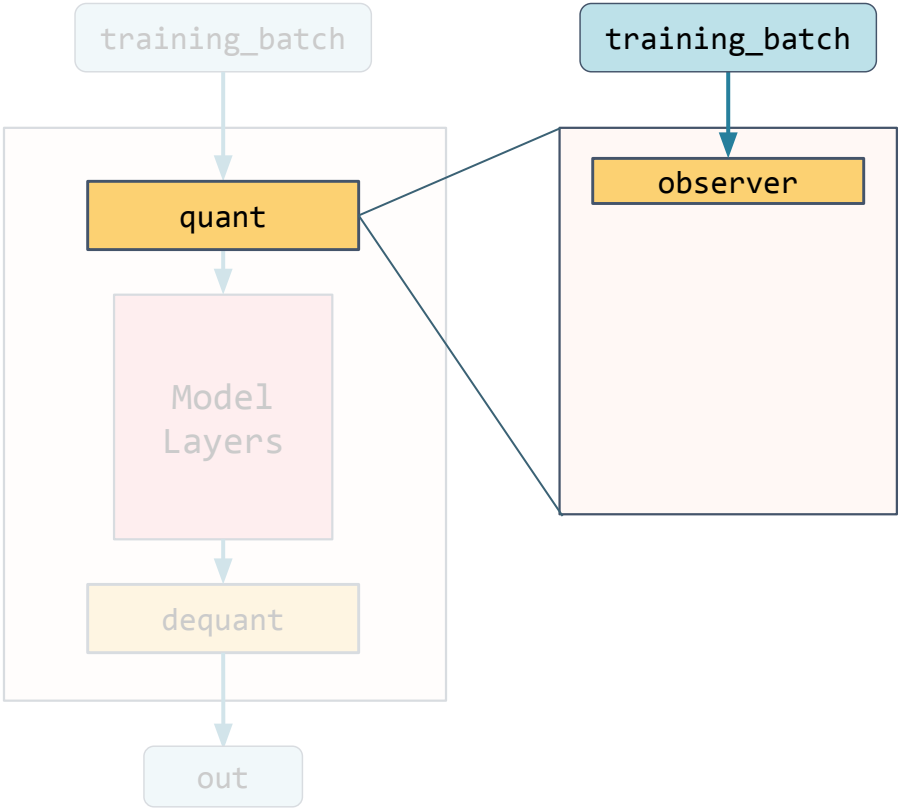
QAT



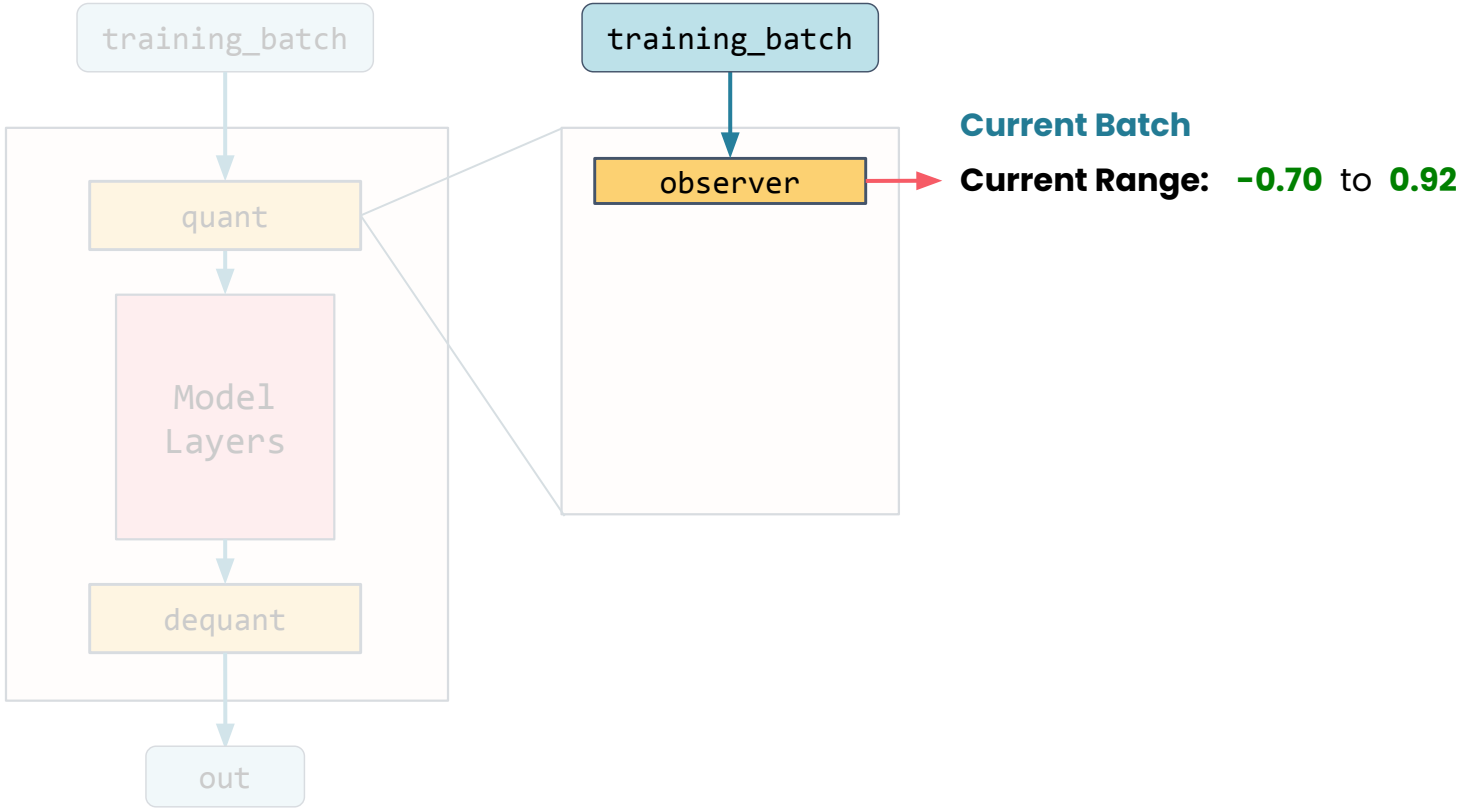
QAT


training_batch
training_batch
quant
Model
Layers
dequant
out

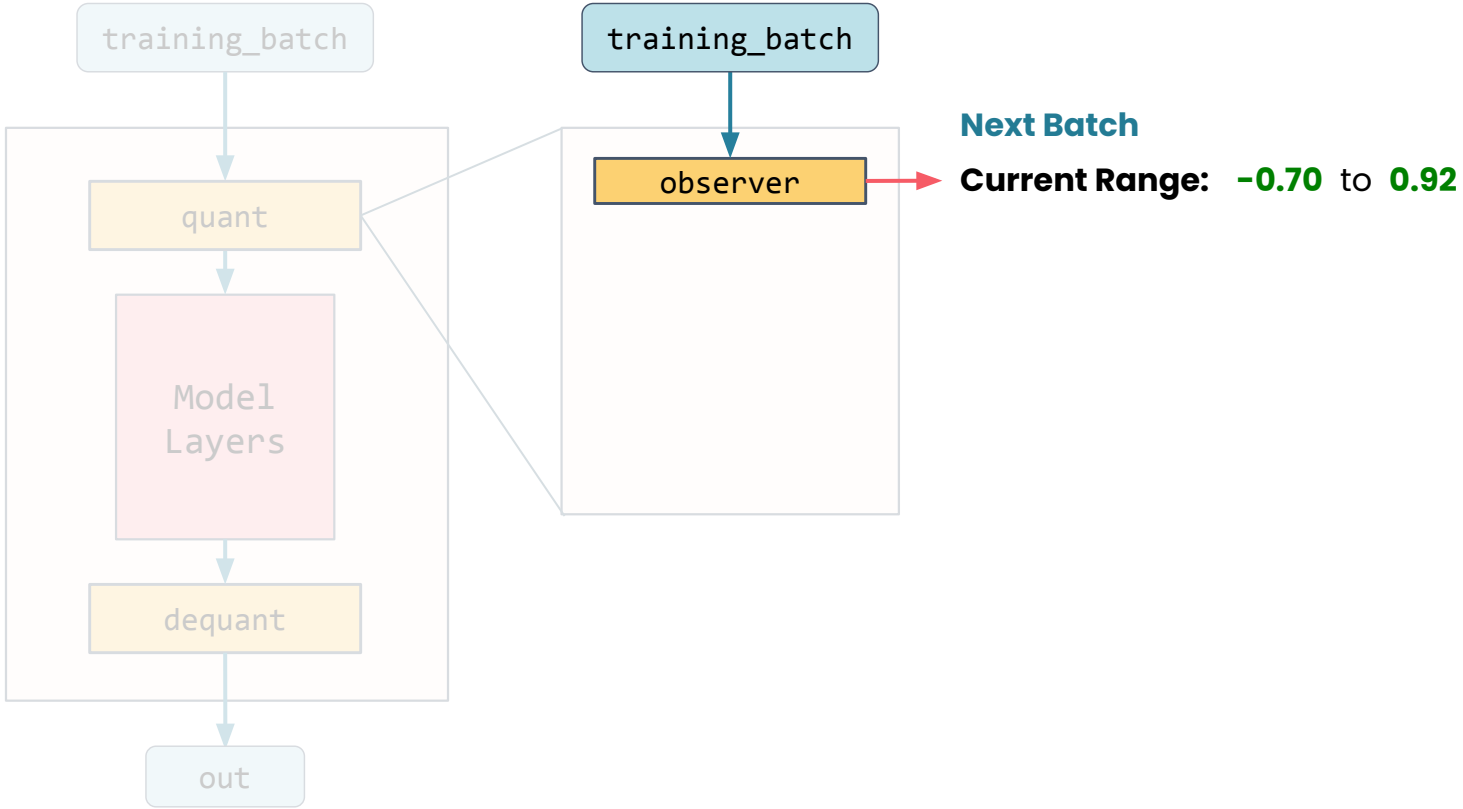
QAT


training_batch
training_batch
observer
quant
Model
Layers
dequant
out

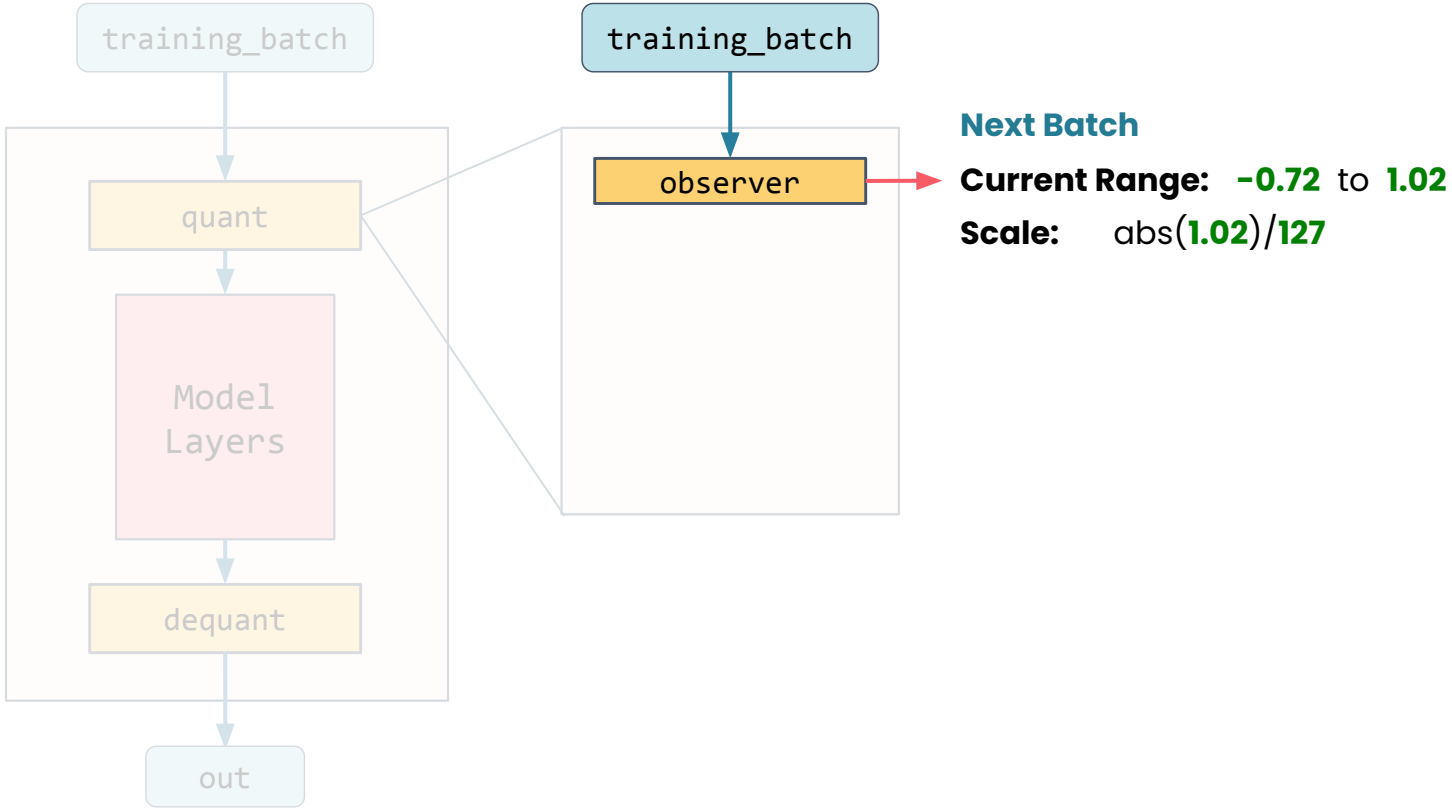
QAT


training_batch
quant
Model Layers
dequant
out
training_batch
observer
Current Batch
Current Range: -0.70 to 0.92

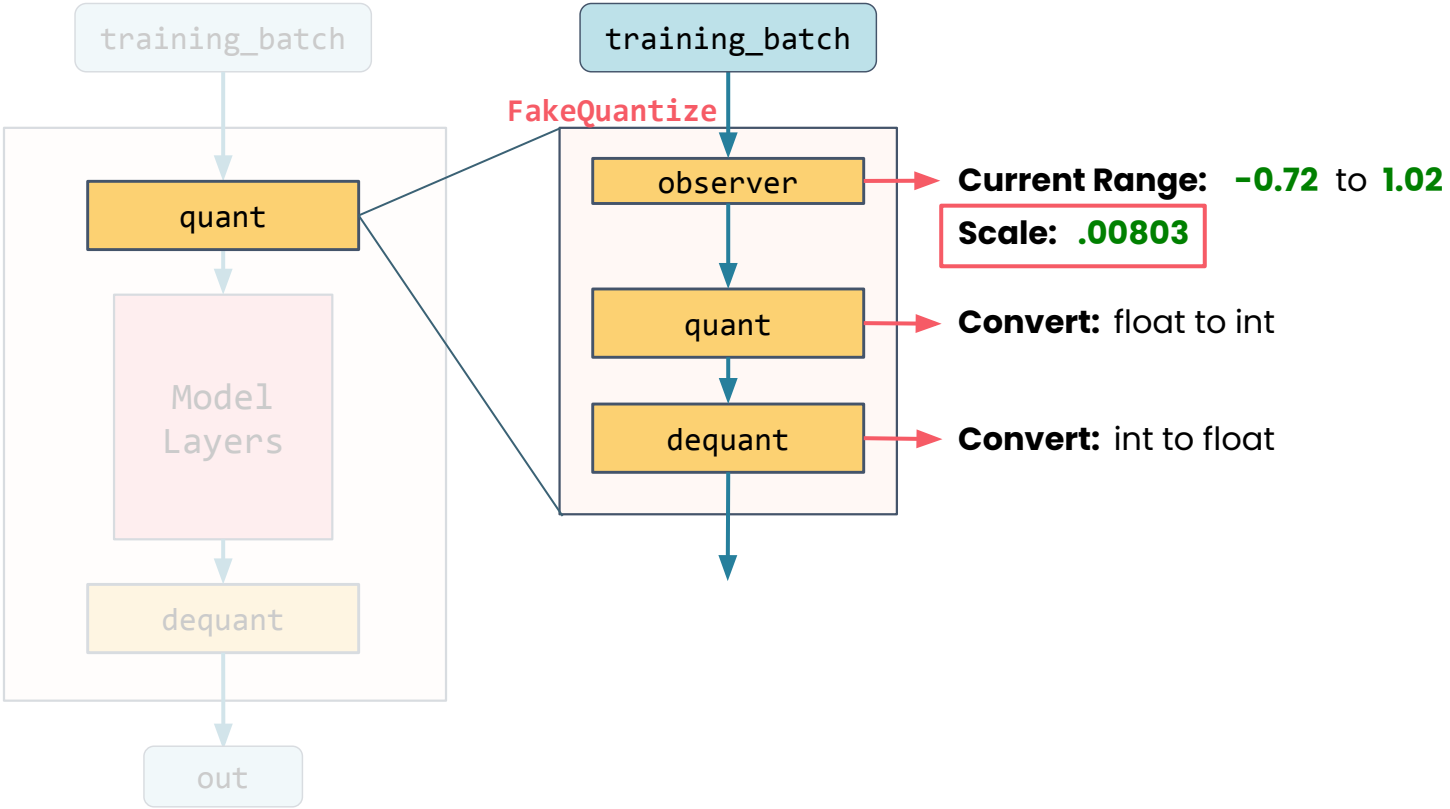
QAT


training_batch
quant
Model Layers
dequant
out
training_batch
observer
Next Batch
Current Range: -0.70 to 0.92

QAT


training_batch
quant
Model Layers
dequant
out
training_batch
observer
Next Batch
Current Range: -0.72 to 1.02
Scale: abs(1.02)/127

QAT


training_batch
quant
Model Layers
dequant
out
training_batch
FakeQuantize
observer
quant
dequant
Current Range: -0.72 to 1.02
Scale: .00803
Convert: float to int
Convert: int to float

Fake Quantization

Activation: **0.125**

Scale: **0.041**

Fake Quantize

Quantize: **round**(**0.125** / **0.041**)

Fake Quantization

Activation: **0.125**

Scale: **0.041**

Fake Quantize

Quantize: **3**

Dequantize: **3*0.041**

Fake Quantization

Activation: **0.125**

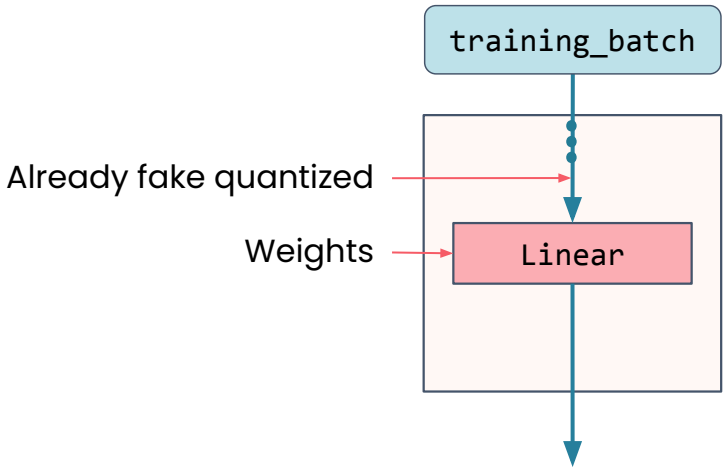
Scale: **0.041**

Fake Quantize

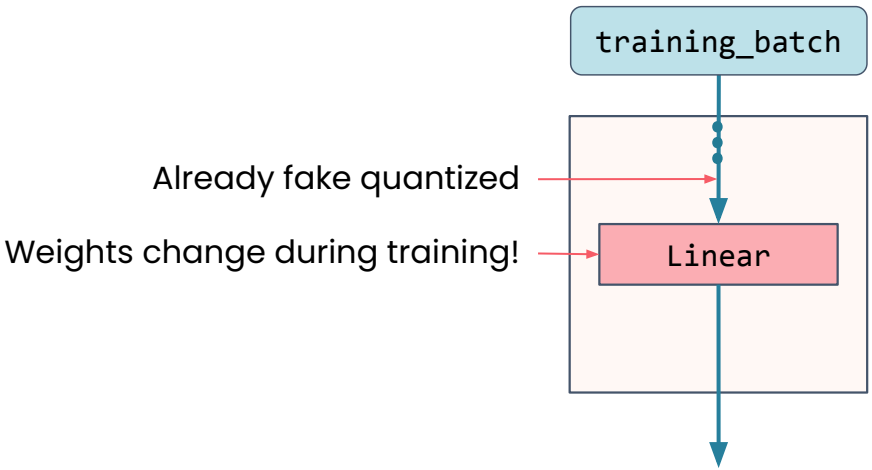
Quantize: **3**

Dequantize: **0.123**

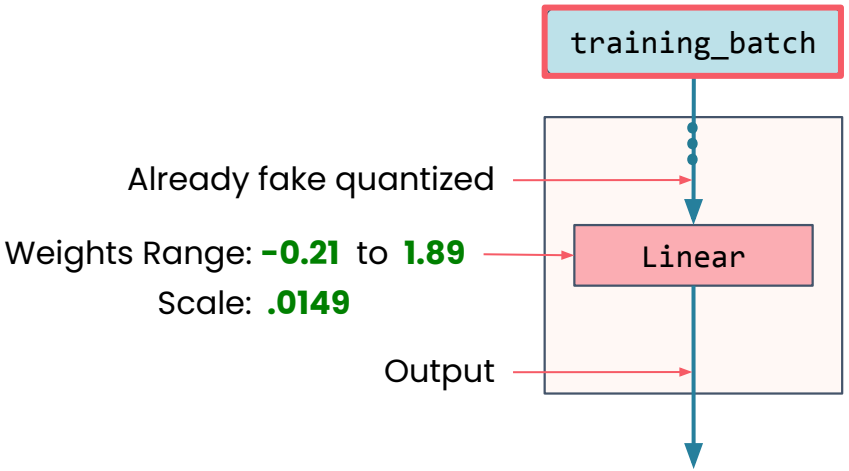
QAT on Layers



QAT on Layers



QAT on Layers



Original weights

```
 0.0283
 1.8984
-0.2156
 1.7618
```

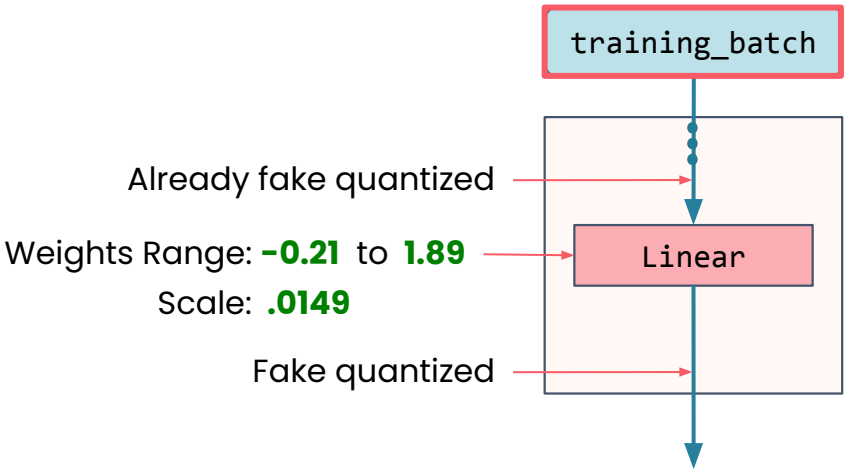
Backprop updates originals in full-precision

Fake Quant Copy

```
 2
 127
-15
 118
```

Used to compute output & loss

QAT on Layers



Original weights

```
 0.0283
 1.8984
-0.2156
 1.7618
```

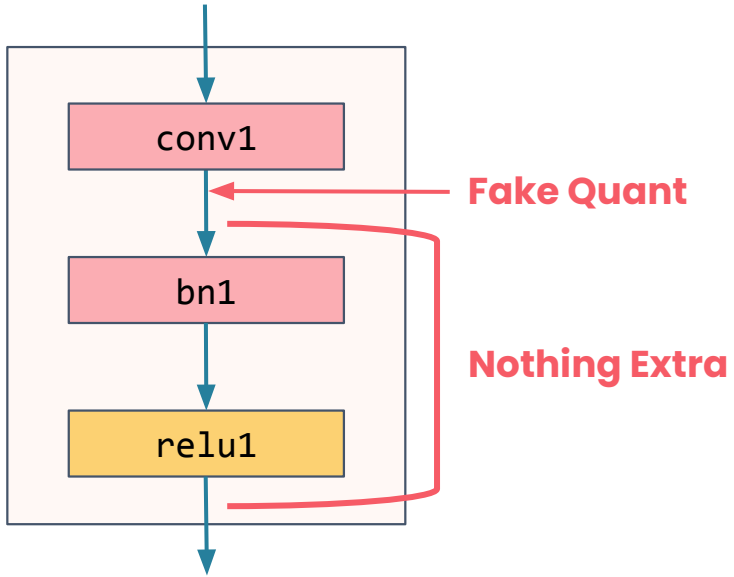
Backprop updates originals in full-precision

Fake Quant Copy

```
 2
 127
-15
 118
```

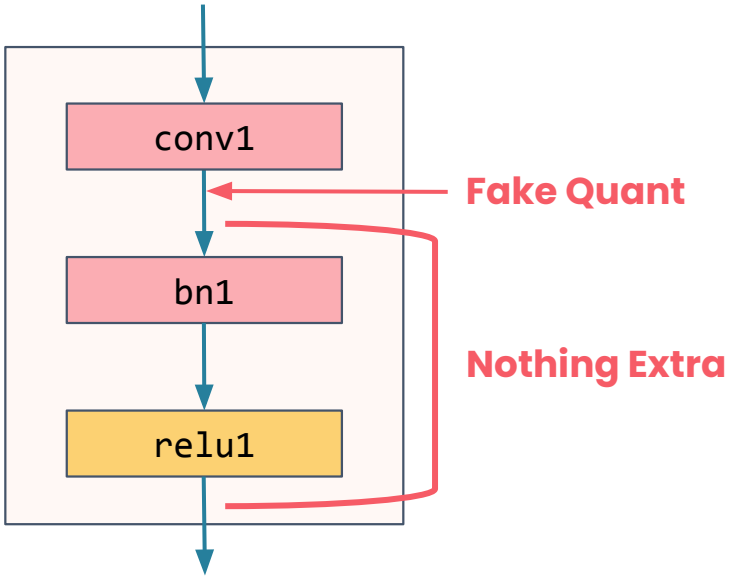
Used to compute output & loss

QAT on Layers



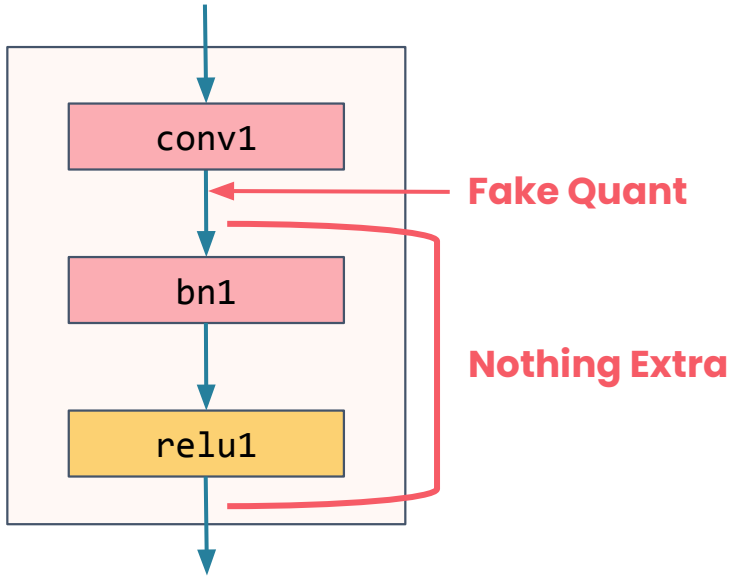
QAT on Layers

```
class QuantizedCNN(nn.Module):

    . . .

    def fuse_model(self):
        torch.quantization.fuse_modules(self,
            ['conv1', 'bn1', 'relu1'], inplace=True)
```



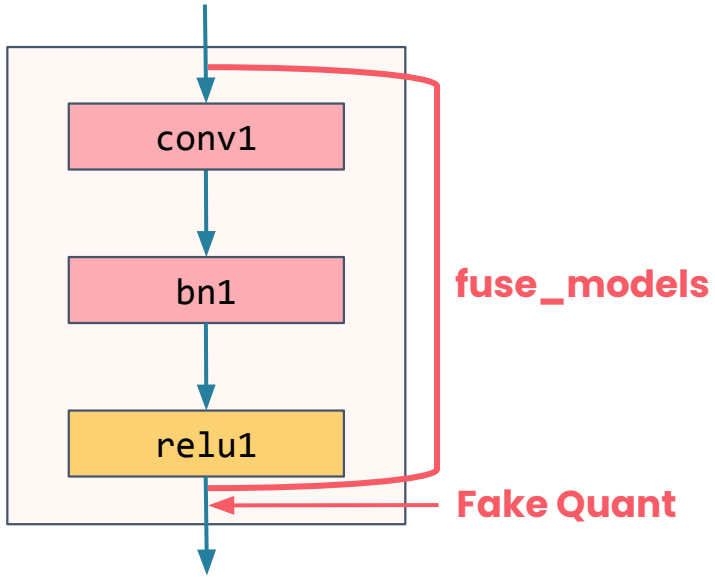
QAT on Layers

```
class QuantizedCNN(nn.Module):

    . . .

    def fuse_model(self):
        torch.quantization.fuse_modules(self,
            ['conv1', 'bn1', 'relu1'], inplace=True)
```



QAT on Layers

```
class QuantizedCNN(nn.Module):

    . . .

    def fuse_model(self):
        torch.quantization.fuse_modules(self,
            ['conv1', 'bn1', 'relu1'], inplace=True)
```



QAT Setup

- **Include stubs**
- **Fuse layers**
- **Load pretrained weights**
- **Prepare and finetune the model**
- **Convert final model**

DeepLearning.AI

PyTorch for Deep Learning

Conclusion