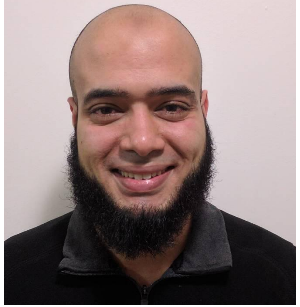
Machine Learning

Misc DNN Topics

Mostafa S. Ibrahim
Teaching, Training and Coaching for more than a decade!

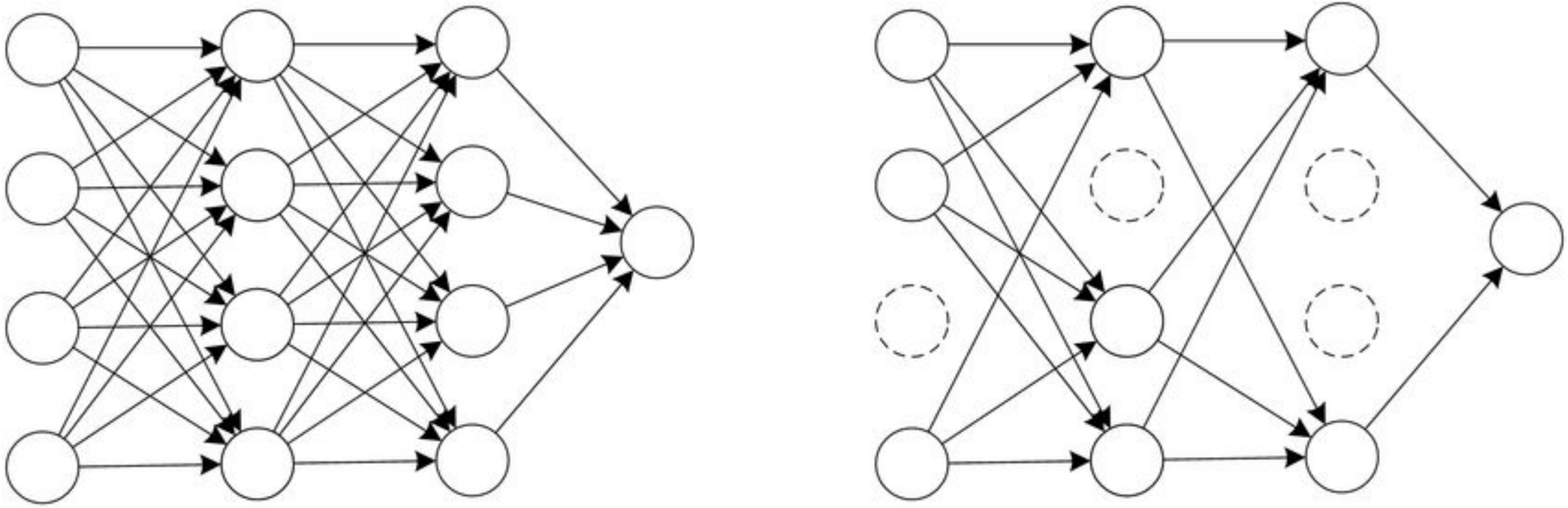
Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / MSc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)



Dropout

- One of the great reguarizers that fits overfitting in deep learning
- It works by randomly "dropping out" a number of neuron outputs in a layer during a training feedforward
  - This process prevents units from **co-adapting** too much to the data
  - The network learns the pattern rather than memorizing using its huge # of weights
  - **self.dropout = nn.Dropout(0.5)**
- It is applied on training only. On testing, all nodes are applied
  - model.eval() is critical
- Where to apply?
  - Mainly In front of linear layers with maybe dropout = 0.5
  - You may explore after some pooling layers with dropout = 0.1 / 0.2
- Video

Dropout



(a) Standard Neural Network (b) Network after Dropout

Img [src]

Batchnorm (2015)

- Observation: distribution of each layer's inputs changes during training
  - Reason: as the parameters of the previous layers change
  - Consequence: slow down the training process and make it harder for the network to converge
- Solution: **Normalize** layer’s input + learn scale/shift **parameters**
  - In **train**: Use batch mean and batch std
  - In **inference**: uses the **entire** training set's **moving average** of the mean and variance
    - Again needs model.eval()
  - Addition side effects: Regularization Effect / Faster Convergence
  - Cons: Dependency on Batch Size / not straightforward for dynamic RNNs
    - Alternatives: Layer Normalization, Instance Normalization, and Group Normalization
- Where to apply?
  - After convolutional layers / fully connected layers but before activation function (debate)
    - Tip: if a batchnorm after conv layer, don’t use bias term in conv (redundant)
  - Resnet blocks

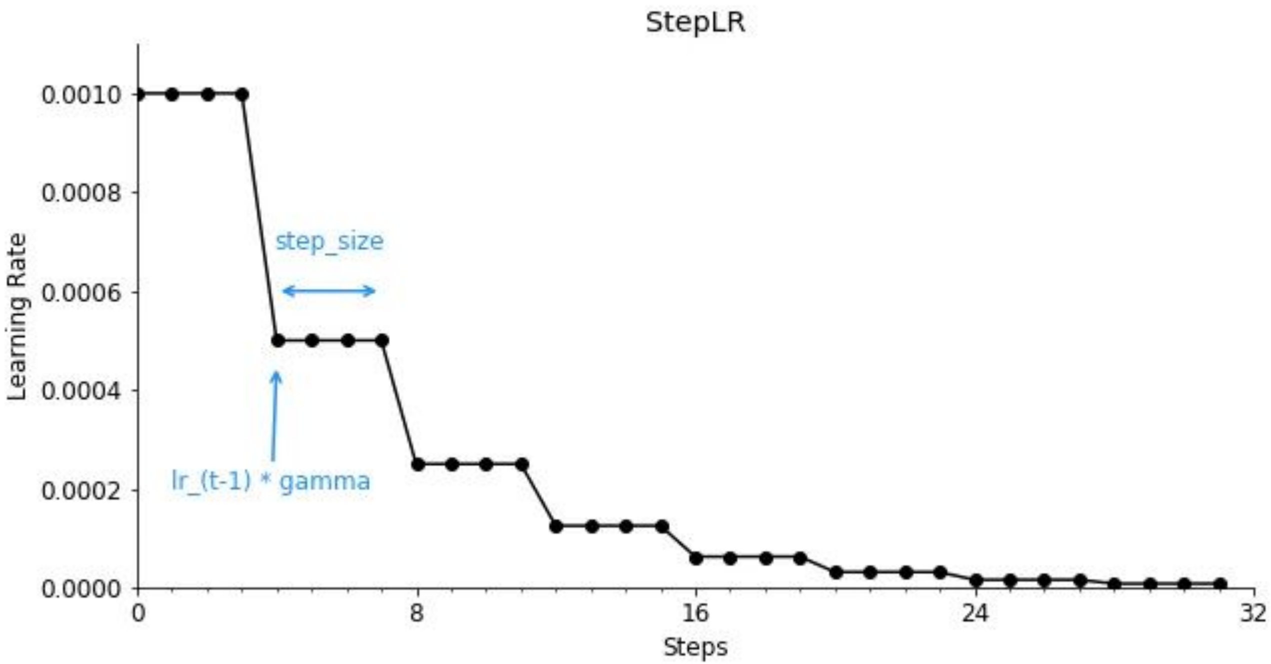
AdamW Optimizer [high level]

- AdamW is an improvement from Adam Optimizer
    - AdamW fixes an implementation mistake about the weight decay
    - It works on really many datasets. Consider as your starting point
- You can think of it as combination of **SGD with momentum** and **RMSProp** optimizers
- Adam maintains **two moving averages** for each parameter
    - At each step, Adam computes **adaptive learning rates for each parameter**
- Videos: Exponentially Weighted Averages / Gradient Descent With Momentum / Adam / Overall

Learning Rate Schedulers [Article]

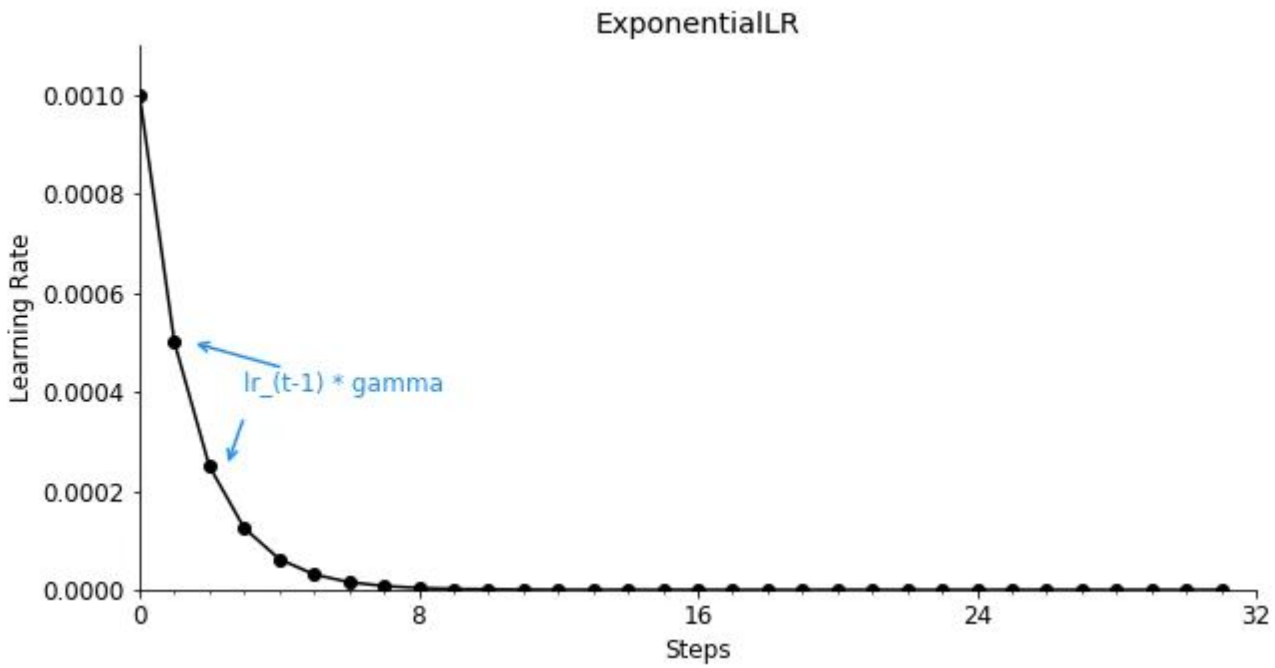
- We don’t just fix one learning rate. We need it to vary during the training
- There are many strategies for that
  - **StepLR**: One of the easiest/oldest ways
    - Decays the learning rate of each parameter group by gamma every step_size epochs.
    - scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=30, gamma=0.1)
    - Then, the gamma and step_size are hyperparameters!
      - Gamma like 1/10 or ½ are common choices
      - Check the loss curve to decide a good place for the step_size
        - MultiStepLR gives more variable milestones [30,80]
  - ExponentialLR: Decays by gamma **every epoch** [common]
  - ReduceLROnPlateau: when a metric has stopped improving
  - CosineAnnealingLR
  - OneCycleLR: read

Learning Rate Schedulers



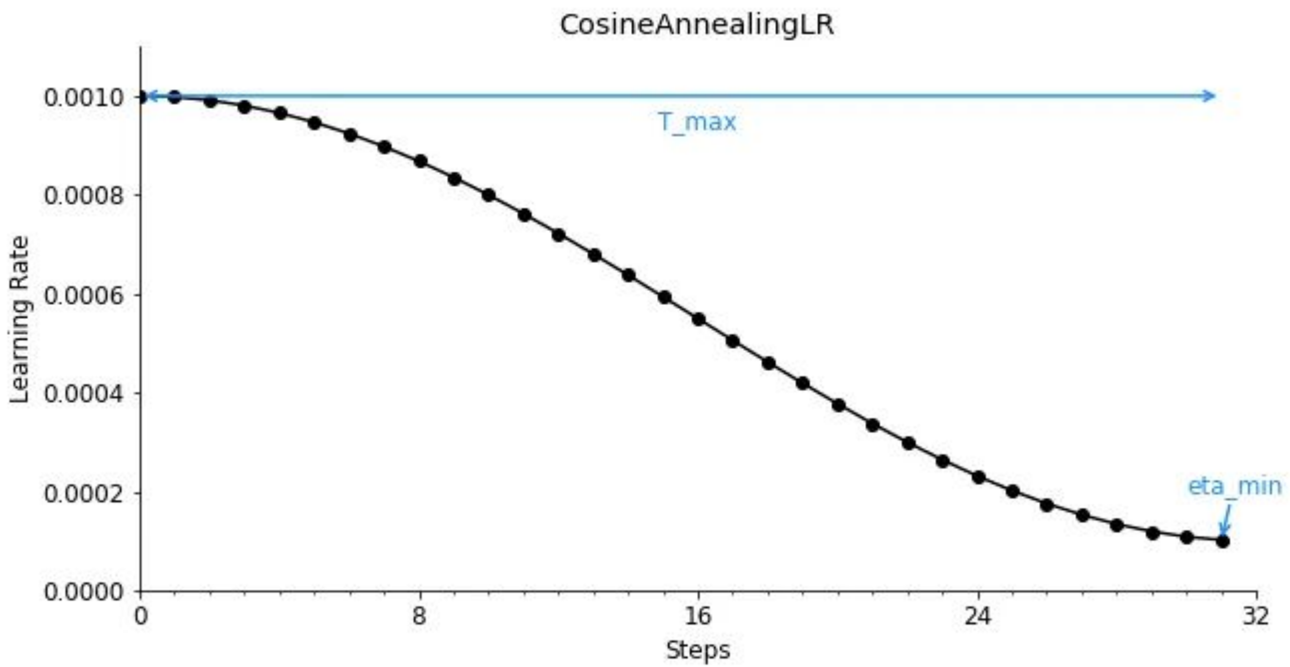
Img [src]

Learning Rate Schedulers



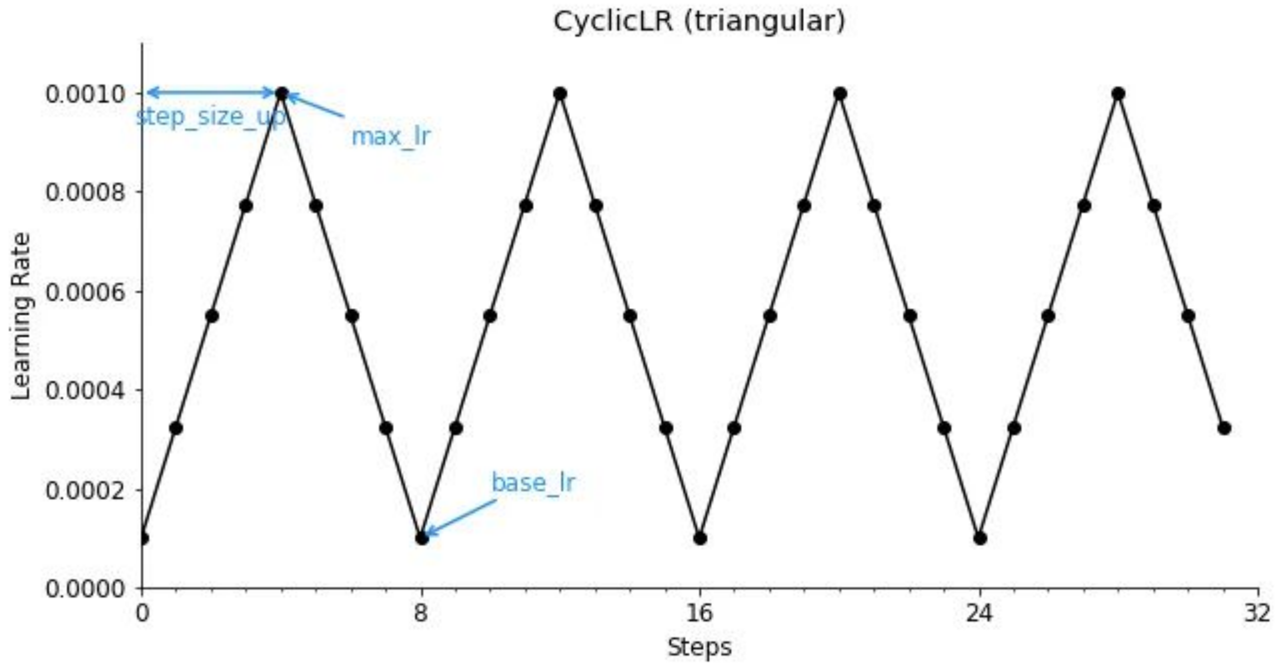
Img src

Learning Rate Schedulers



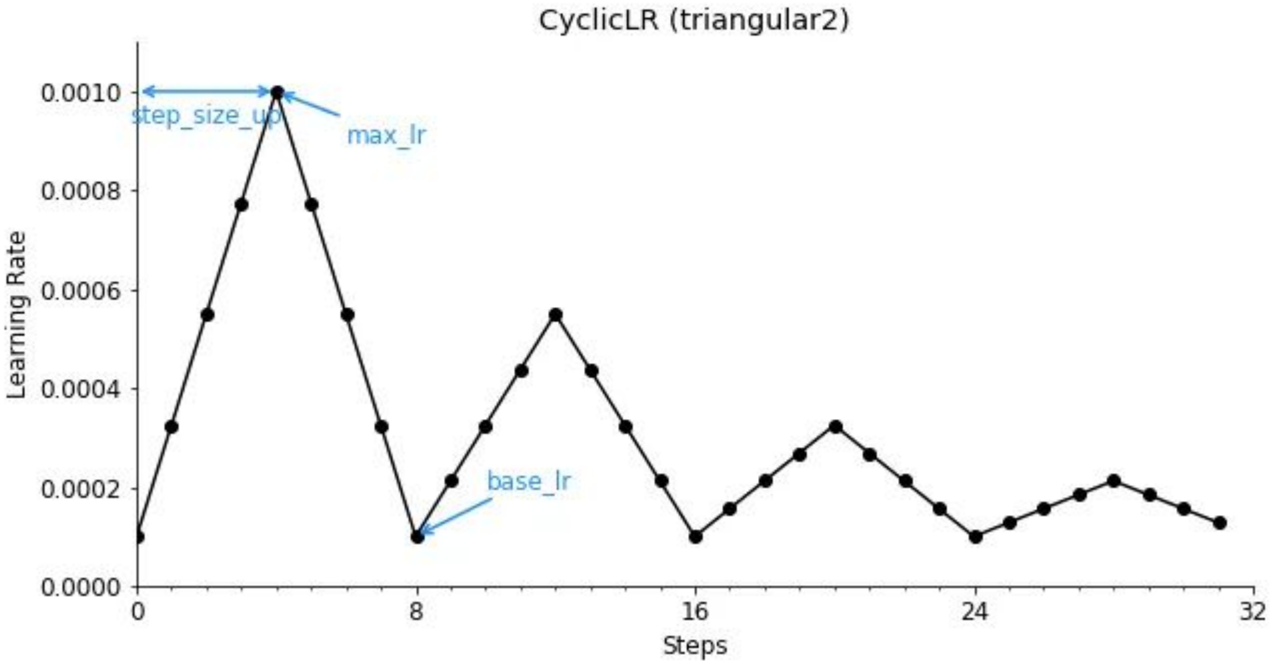
Img [src]

Learning Rate Schedulers



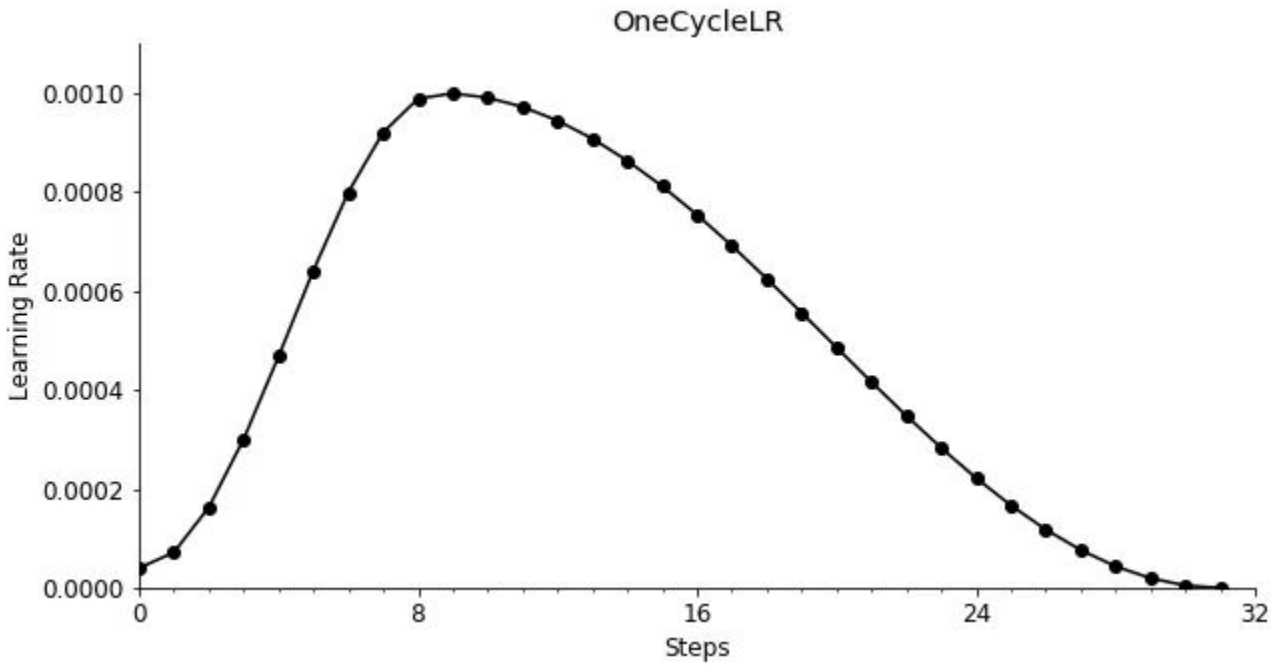
Img src

Learning Rate Schedulers



Img [src]

Learning Rate Schedulers



Img src

Vanishing Gradients

- During backpropagation in deep networks the gradients become very small the more we move backward in the hidden layers
  - As a result no or very slow learning as the gradient signal is too weak
  - The opposite is called exploding gradients
- Causes
  - **Improper initialization**: In feedforward, we are **multiplying weights** from the matrices
  - **Activation Functions**: In backpropagation, with chain rule, we **multiply gradients**
    - Sigmoid and tanh activations become extremely small for small/large inputs
  - The deeper the network, the harder to train due to these 2 reasons
    - Similarly long sequences in RNNs

Vanishing Gradients

- A good solution must tackle all these issues
  - **Weight Initialization**: Using He initialization or Xavier initialization
  - **Activation Functions**: Relu (+ve range derivative = 1) / non-saturating function
    - A saturating function: approach a plateau as the input grows large (sigmoid/tanh)
  - **Skip connections** (like in ResNets) allow gradients to bypass layers
  - **Batch Normalization**: Normalizing the input of each layer to have mean=0, variance=1
  - **Gradient Clipping**: Put a range to clip
- For exploding gradients, in addition to above solutions, **weight regularization** and **lower learning rate** are used

Network Initialization

- Initializing the network is a very critical component in training DNNs
- Early days, people realized some lessons
  - Very small weights leads to vanishing gradients and large ones to exploding gradients
  - Weights can’t be the same constant. They should be different and have variance
  - Normal and Uniform distribution are good ways
- However, this was not that enough with deep learning
  - Weight initialization methods need to be compatible with the choice of an **activation function**, mismatch can potentially affect training negatively.
  - Just random initialize the weight may cause vanishing/exploding gradient

Network Initialization

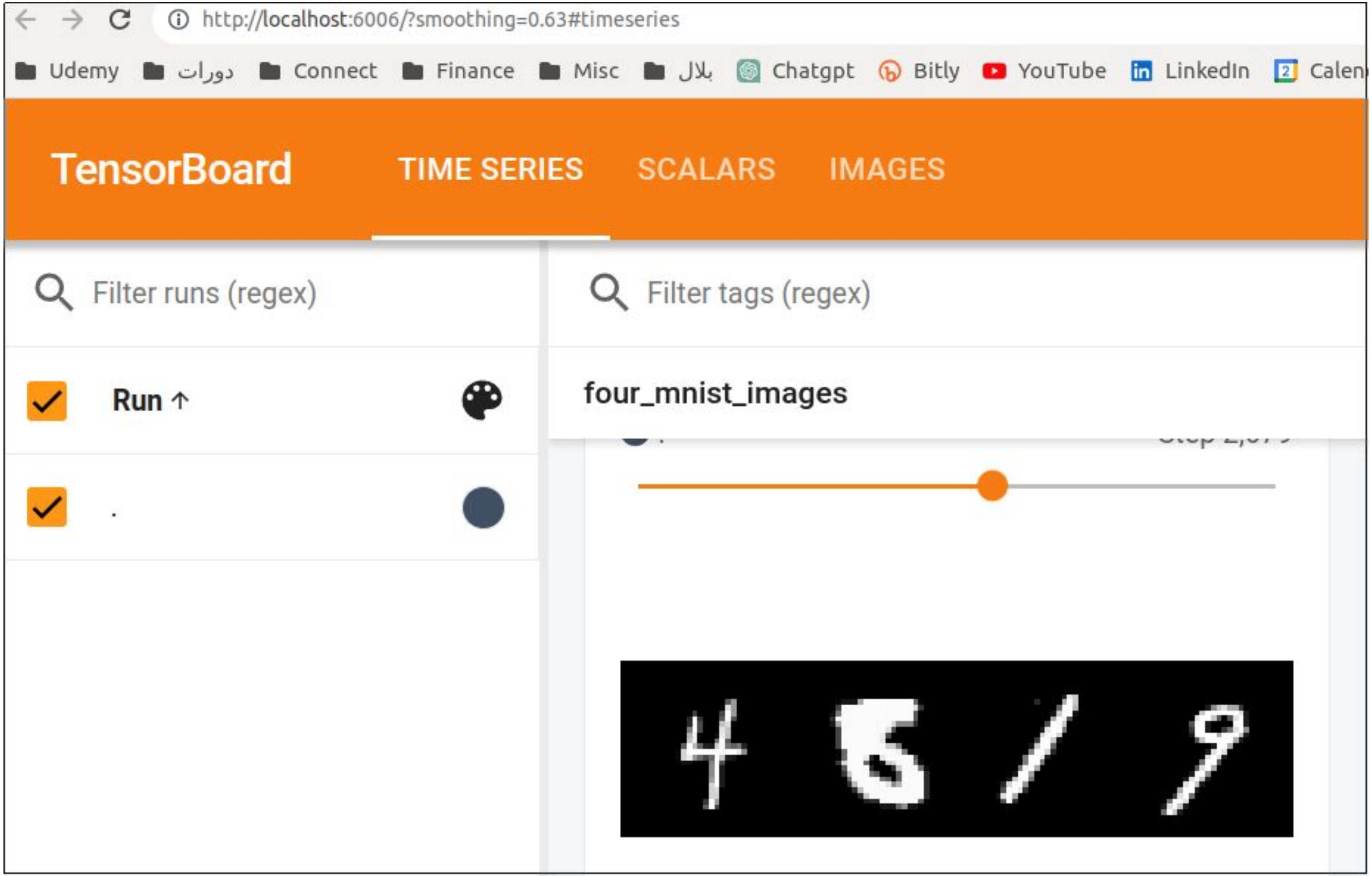
- There are 2 major techniques nowadays to tackle deep learning
- **Xavier**/Glorot initialization [paper]
  - "Adjust the scale of the initial weights based on the number of **input and output** neurons in a way that aims to keep the **variance** of the activations **constant** across layers."
  - Some say it is good for tanh/sigmoid activations, but TF uses as default / good for DNN
- **Kaiming**/He initialization
  - Designed specifically for **Deep Networks with RELU**: paper Delving Deep into Rectifiers
  - Most PyTorch layers use **Kaiming with Uniform distribution** for most of the layers
    - Such as Conv, Linear and RNN
    - However, the bias is sampled from the uniform distribution
    - These distributions are sampled from a **specific range** based on tensor input/output

Network Initialization: Custom

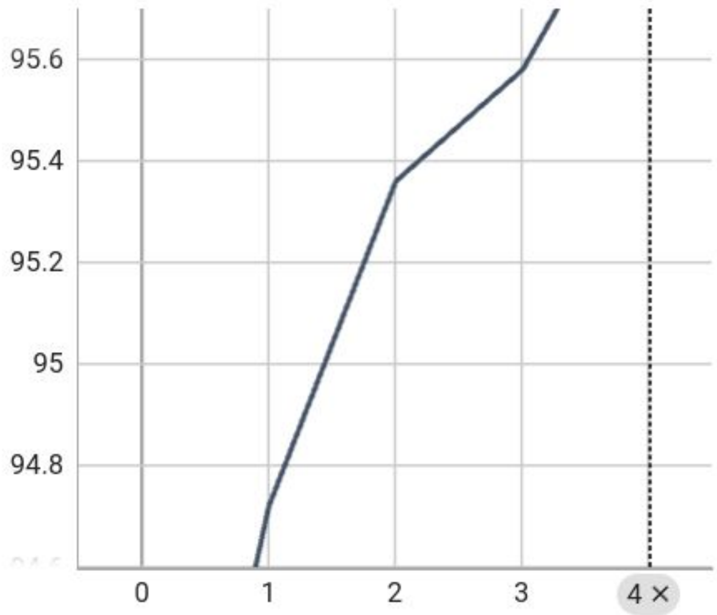
```python
def init_weights(m):
    if isinstance(m, nn.Conv2d):
        init.xavier_uniform_(m.weight)
        if m.bias is not None:
            init.zeros_(m.bias)
    elif isinstance(m, nn.BatchNorm2d):
        init.constant_(m.weight, 1)
        init.constant_(m.bias, 0)
    elif isinstance(m, nn.Linear):
        init.kaiming_normal_(m.weight)
        if m.bias is not None:
            init.zeros_(m.bias)


model = nn.Sequential(
    nn.Conv2d(1, 20, 5),
    nn.ReLU(),
    nn.Conv2d(20, 64, 5),
    nn.ReLU(),
    nn.Linear(64, 10)
)

model.apply(init_weights)
```

Tensorboard Visualization

- TensorBoard is a **visualization toolkit** (web app) for machine learning experimentation inspection developed by the TensorFlow team.
    - Adopted in PyTorch and other machine learning frameworks
- You can
    - Track and visualize **metrics** such as loss and accuracy
    - Visualize the model graph
    - Display **images**, text, and audio data samples
    - View the **distribution** of weights, biases, or other tensors as they change over time.
    - Track hyperparameter tuning sessions using the HParams dashboard.
        - from tensorboard.plugins.hparams import api as hp / writer.add_hparams()
- Install from: **pip install tensorboard**
- Then build and visualize web from: **tensorboard --logdir=<Path>**

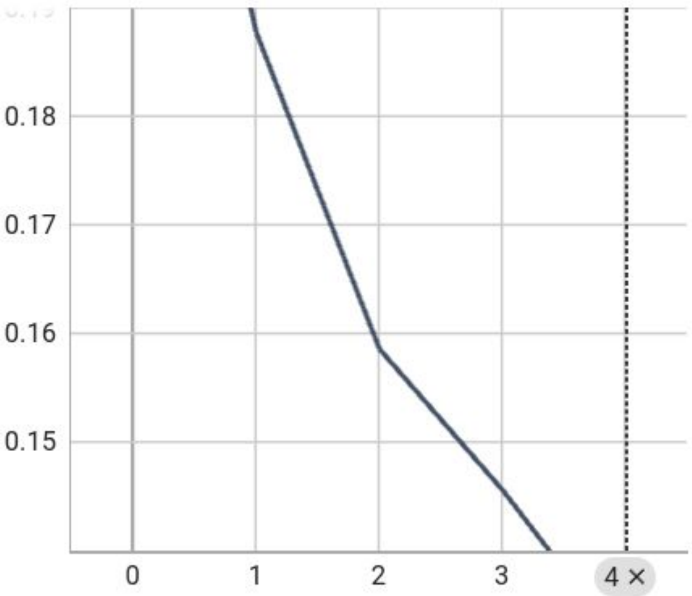

http://localhost:6006/?smoothing=0.63#timeseries
Udemy
دورات
Connect
Finance
Misc
بلال
Chatgpt
Bitly
YouTube
LinkedIn
TensorBoard
TIME SERIES
SCALARS
IMAGES
Filter runs (regex)
Filter tags (regex)
Run
four_mnist_images

validation_accuracy



Run ↑	Value	Step	Relative
.	96.02	4	24.92 sec

validation_loss



Run ↑	Value	Step	Relative
.	0.1308	4	24.92 sec

```python
writer = SummaryWriter('runs/mnist_experiment_1')
```

```python
# Log the running loss averaged per mini-batch
writer.add_scalar('training_loss', running_loss / 100, epoch * len(train_loader) + i)

# Log a random batch of images
img_grid = torchvision.utils.make_grid(inputs[:4].cpu().data)
writer.add_image('four_mnist_images', img_grid, epoch * len(train_loader) + i)
```

```python
        # Log validation loss and accuracy
        writer.add_scalar('validation_loss', avg_val_loss, epoch)
        writer.add_scalar('validation_accuracy', val_accuracy, epoch)

    print('Finished Training')

# Call the training loop
train(num_epochs=5)

# Close the TensorBoard writer
writer.close()
```

- See, run and visualize the full code

What you need in a good project?

- Data Loader
  - Always start with data loader and verify the data properly
  - Prepare sample mode (for fast check / debugging) and full mode loadings
- Configuration
  - Any hyperparameter or model choices must be from a configuration file
- Versioning
  - You must be able to know which code/version/data generated these results
  - For example, git code version + copy config files at minimum
- Logger
  - You must log every important thing such as model layers, data path, configuration content
- Events
  - Save events for later tensorboard visualization
- Multi-GPU support / train/eval round strategy

Google Colab

- Colab is a free, cloud-based service that supports ML
  - Write and execute Python in your browser
  - GPUs and TPUs for intensive computational tasks.
  - Integration with Google Drive: save and share your work
  - Supports TensorFlow, PyTorch, Keras, and OpenCV
  - Easy to use for collaboration
  - No setup

Relevant Materials

- Weight Initialization in Neural Networks
- Weight Initialization Techniques-What best works for you
  - Focus on the text not the equations
- What is default weight and bias initialization in PyTorch?

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."