Additional Techniques for Training Neural Networks

Sourav Karmakar

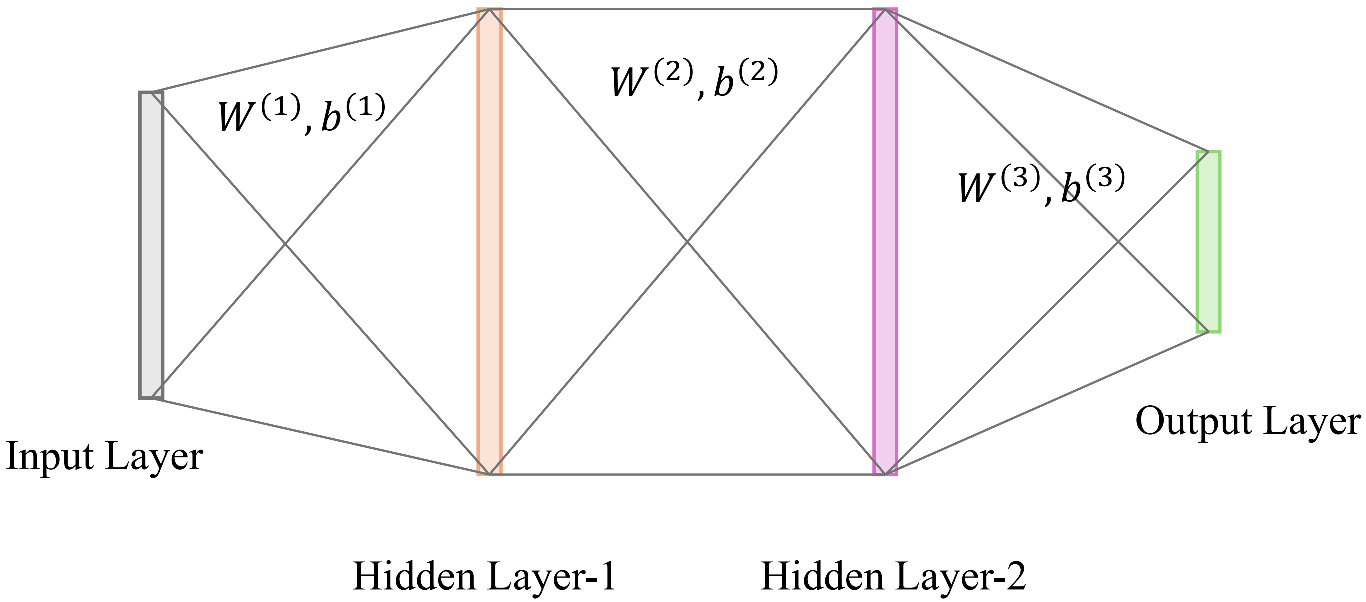
souravkarmakar29@gmail.com

Section-1: Regularization

Why Regularization is needed in NN?

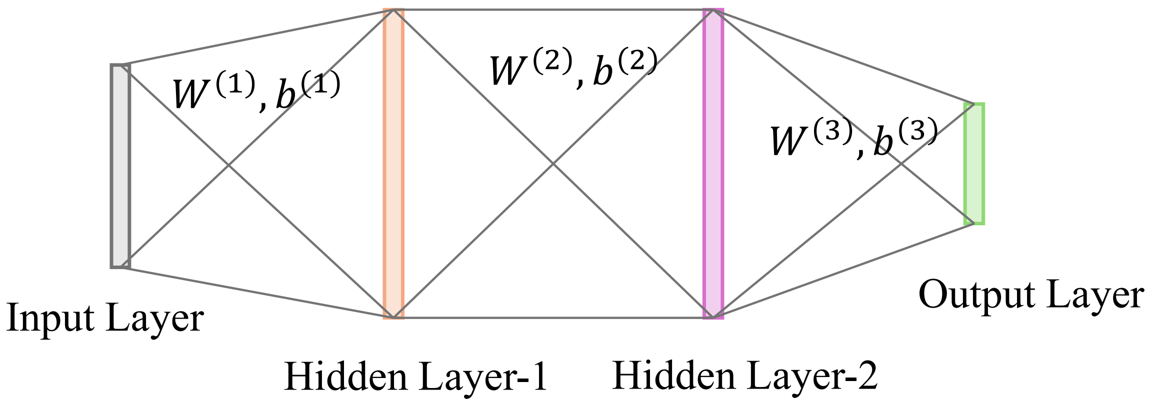
Imagine a Neural Network (MLP) with 50 input features and 10 class labels. Suppose, we have two hidden layers each with size = 100 (i.e. 100 hidden nodes in each hidden layers). How many parameters does this NN have?

$W^{(1)}$: weight of the first linear layer ; $b^{(1)}$: bias of the first linear layer
$W^{(2)}$: weight of the second linear layer ; $b^{(2)}$: bias of the second linear layer
$W^{(3)}$: weight of the third linear layer ; $b^{(3)}$: bias of the third linear layer



Why Regularization is needed in NN?

Imagine a Neural Network (MLP) with 50 input features and 10 class labels. Suppose, we have two hidden layers each with size = 100 (i.e. 100 hidden nodes in each hidden layers). How many parameters does this NN have?



Size of $W^{(1)} = size\ of\ first\ hidden\ layer \times input\ size = 50 \times 100 = 5{,}000$;
Size of $b^{(1)} = size\ of\ first\ hidden\ layer = 100$

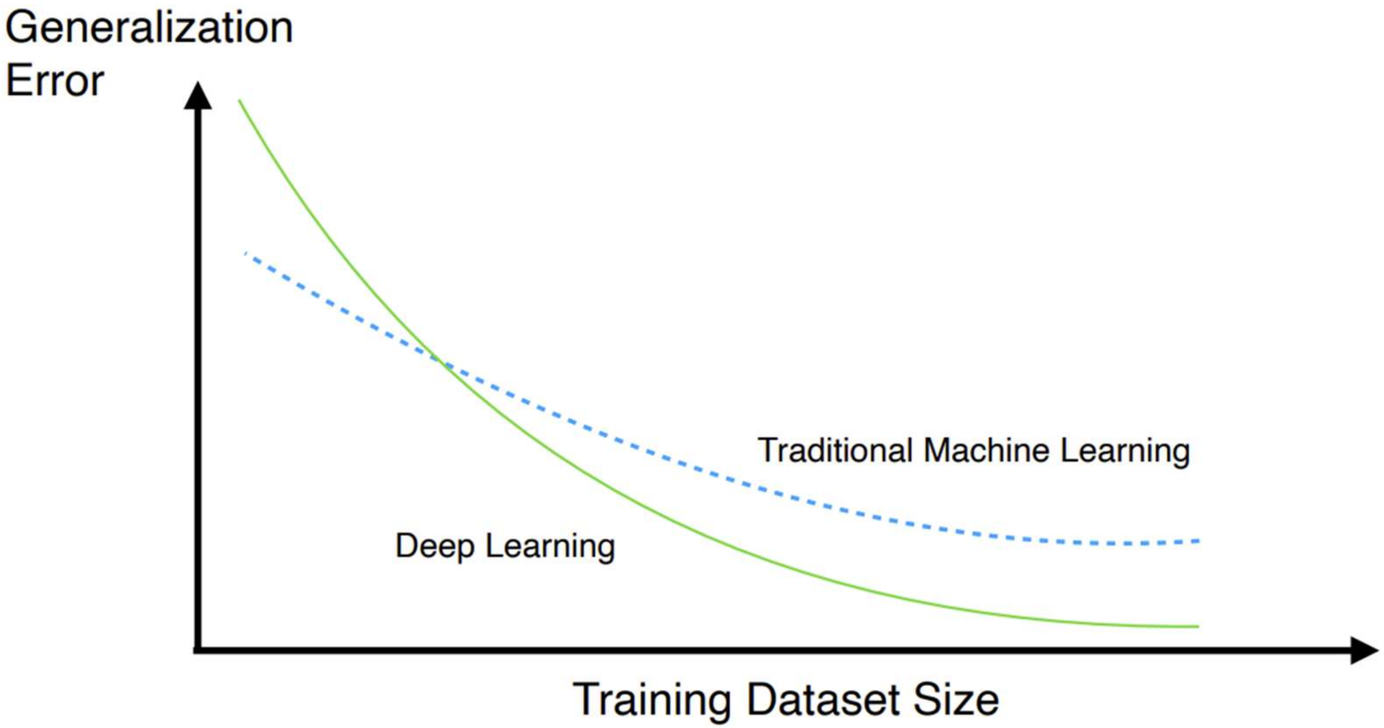
Size of $W^{(2)} = size\ of\ second\ hidden\ layer \times size\ of\ first\ hidden\ layer = 100 \times 100 = 10{,}000$;
Size of $b^{(2)} = size\ of\ second\ hidden\ layer = 100$

Size of $W^{(3)} = size\ of\ output\ layer \times size\ of\ second\ hidden\ layer = 10 \times 100 = 1000$;
Size of $b^{(3)} = size\ of\ output\ layer = 10$

Total number of parameters: (5000+100) + (10000+100) + (1000+10) = 16210

Why Regularization is needed in NN?

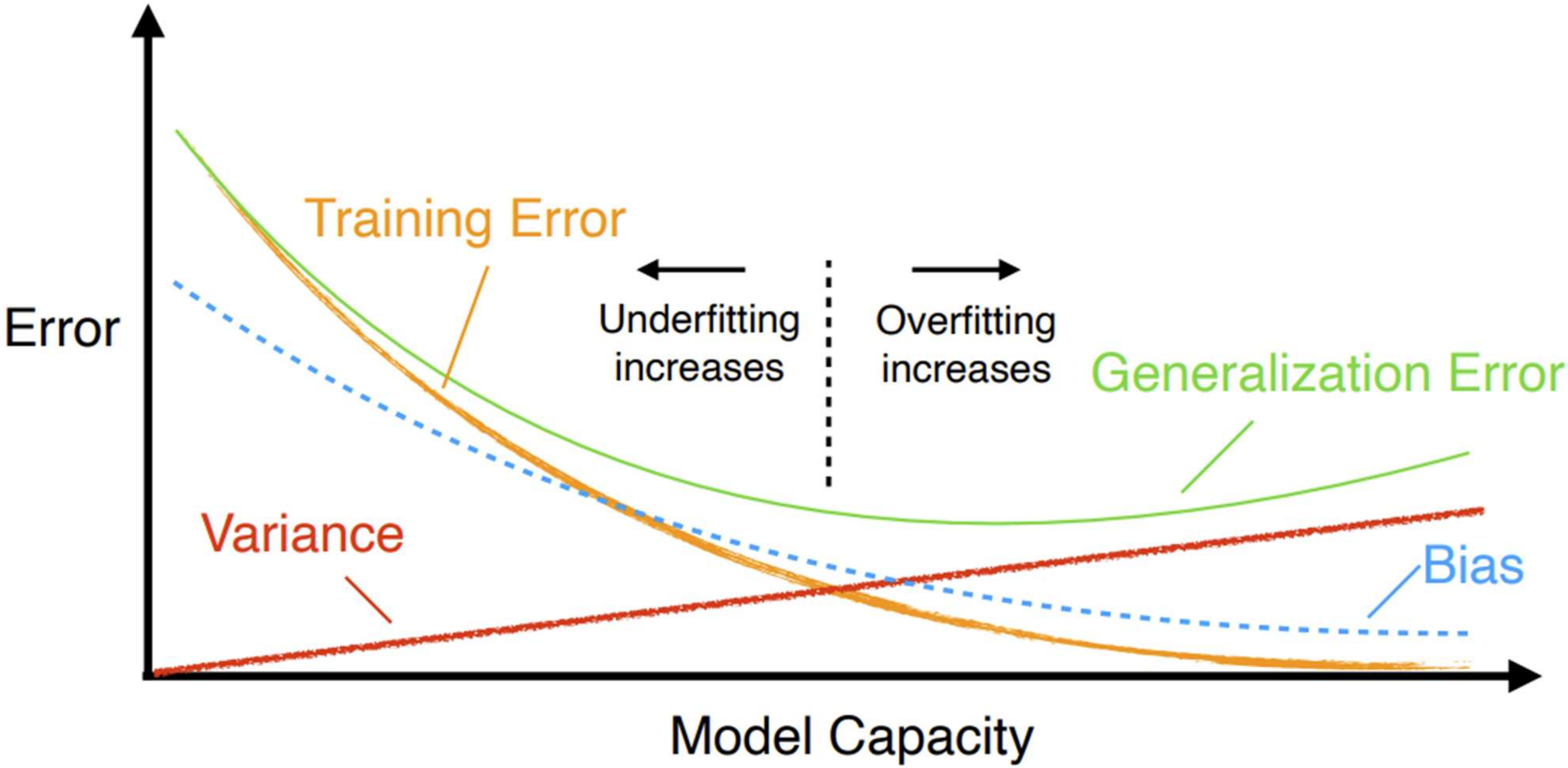
So, our multi-layer neural network contains a lots of parameters. Having a lot of parameter means the model will need huge amount of data to train properly. i.e. why deep learning architectures (i.e. neural nets with multiple layers) works best with huge amount of data.



What if we do not have enough training data or the model is too complex?

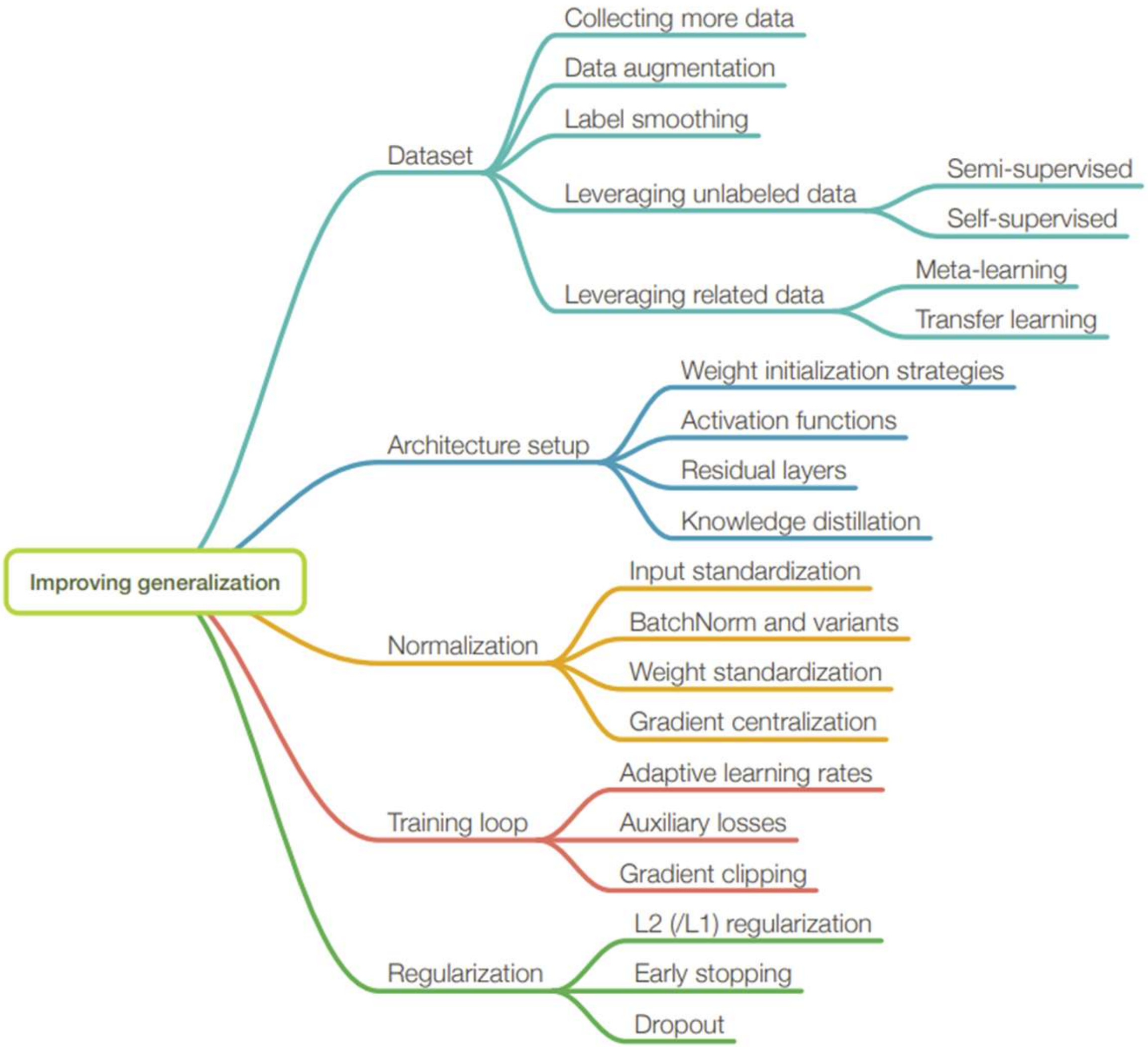
Why Regularization is needed in NN?

The old evil: Bias-Variance Trade off



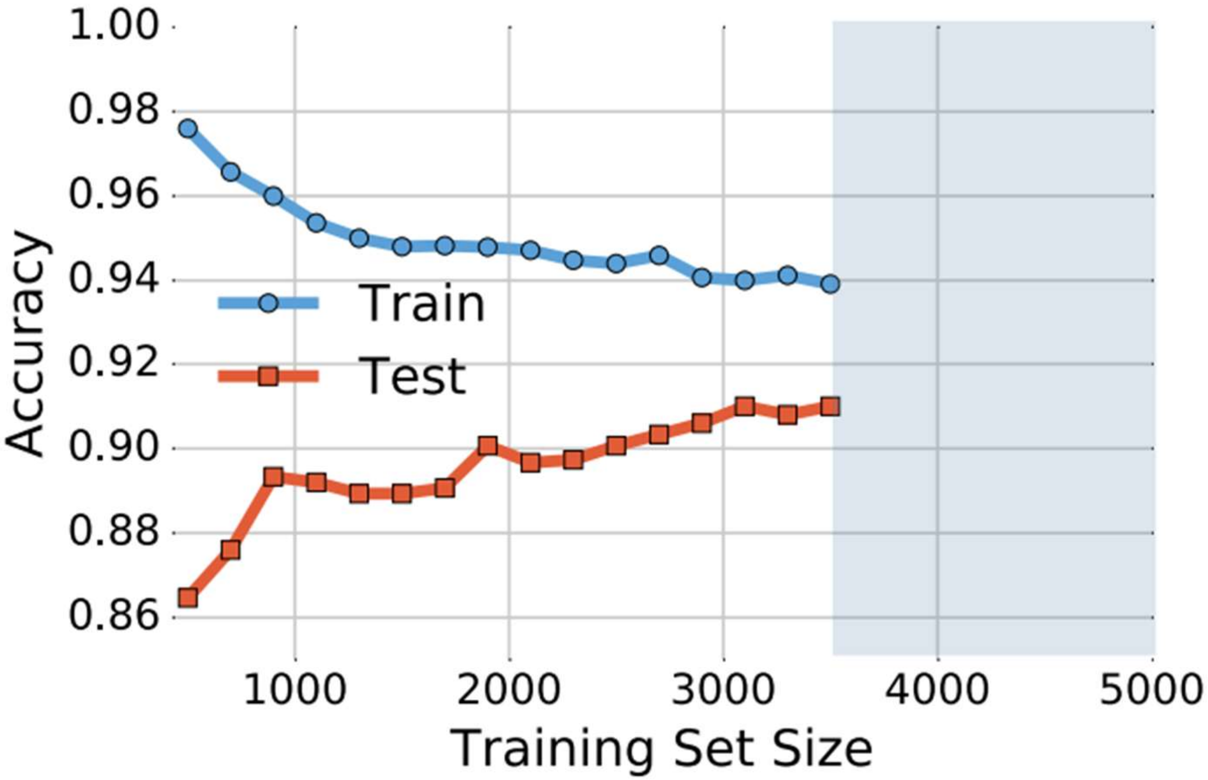
Capacity: abstract concepts ≈ number of model parameters × how efficiently the parameters are used

Improving generalization performance



Avoid overfitting with more data

Often, the best way to reduce overfitting is collecting more data



Softmax on MNIST subset (test set size is kept constant)

However, collecting more data in practice

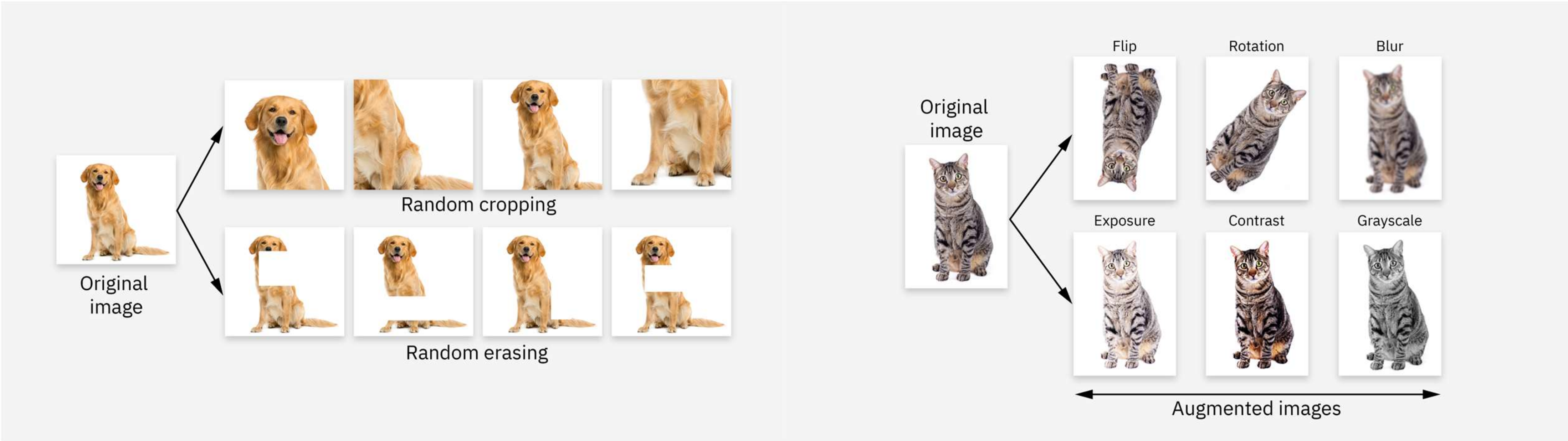
- Requires more time to gather the data.
- More cost involved in gathering the data.
- Can have other constraints.

Hence, researchers and practitioners often tries to come up with strategies to augment the existing dataset.

Data augmentation

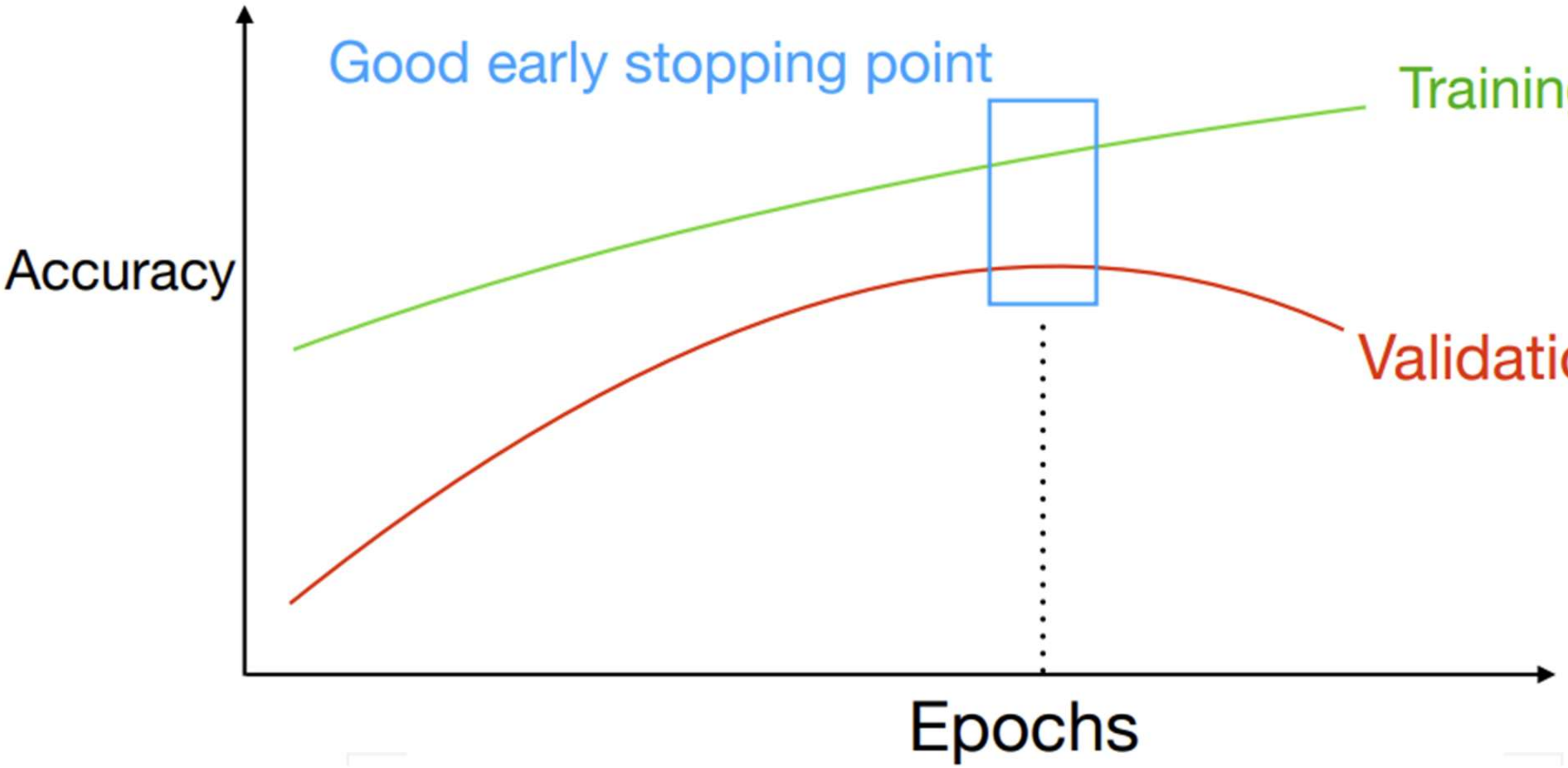
Sometimes, we can enlarge the training dataset by adding little variations of the original dataset. This is called **data augmentation.**

This method is mostly popular in image dataset (computer vision) or sometimes in texts. But for tabular dataset this method doesn't work very well.



Early Stopping

Reduce overfitting by observing the training / validation accuracy gap during training and then stop at the “right” point.



Early stopping is not very common these days.

Early Stopping

```python
best_val_loss = float("inf")
patience, counter = 3, 0  # stop if no improvement for 3 epochs

for epoch in range(100):
    # ---- Training ----
    model.train()
    for X, y in train_loader:
        optimizer.zero_grad()
        loss = criterion(model(X), y)
        loss.backward()
        optimizer.step()

    # ---- Validation ----
    model.eval()
    val_loss = 0
    with torch.no_grad():
        for X, y in val_loader:
            val_loss += criterion(model(X), y).item()
    val_loss /= len(val_loader)

    print(f"Epoch {epoch+1}, Val Loss: {val_loss:.4f}")

    # ---- Early stopping ----
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        torch.save(model.state_dict(), "best_model.pt")
        counter = 0
    else:
        counter += 1
        if counter >= patience:
            print("Early stopping triggered!")
            break

# Load best model
model.load_state_dict(torch.load("best_model.pt"))
```

Define the early stopping criteria

Keep track of validation loss

If validation loss is less than best validation loss, we will set `best_val_loss = val_loss` and reset the counter to zero

If validation loss doesn't improve for 3 epochs, then we will stop model training.

Adding L2 penalties to the loss

$$L_2 - Regularized\ Cost\ (W,b) = \underbrace{\boxed{\frac{1}{n}\sum_{i=1}^{n}\mathcal{L}\left(y^{[i]}, \hat{y}^{[i]}\right)}}_{\text{Loss function}} + \underbrace{\boxed{\frac{\lambda}{n}\sum_{l=1}^{L}\left\|W^{(l)}\right\|_F^2}}_{L_2 \text{ regularization term}}$$

$$Where, \left\|W^{(l)}\right\|_F^2 \ is\ the\ squared\ Frobenius\ Norm\ of\ the\ Weight\ Matrix\ of\ layer - l = \sum_i \sum_j \left(w_{i,j}^{(l)}\right)^2$$

λ is the regularization parameter.

L_2 regularization is also called **Thikonov regularization**. It performs "weight shrinkage" or penalty against complexity.

```
# regularize loss
L2 = 0.
for name, p in model.named_parameters():
    if 'weight' in name:
        L2 = L2 + (p**2).sum()

cost = cost + 1. /targets.size(0) * LAMBDA * L2

optimizer.zero_grad()
cost.backward()
```

```
optimizer = torch.optim.SGD(model.parameters(),
                            lr=0.01,
                            weight_decay=0.05)
```

`weight_decay` is the L_2 regularization factor

Dropout

Dropout in a Nutshell: dropping nodes

How do we drop nodes practically / efficiently?

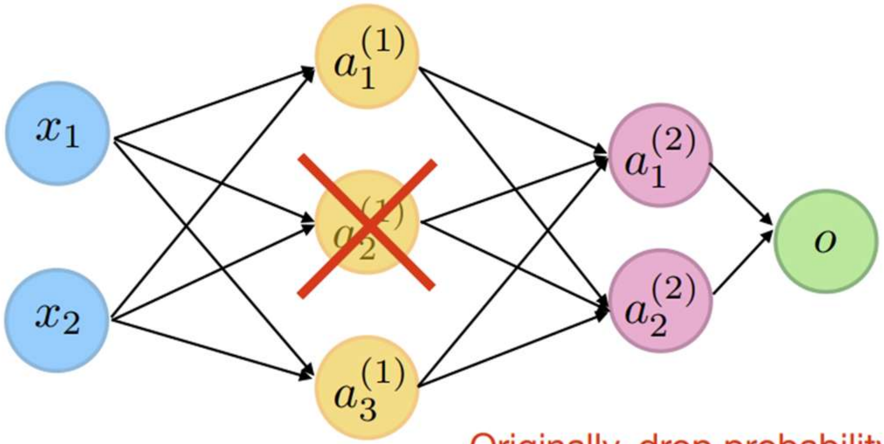
Bernoulli Sampling (during training):

- p $\coloneqq$ drop probability, usually 0.5 but any value between 0.2 to 0.8 works well in practice.
- $\boldsymbol{v}$ $\coloneqq$ random sample from uniform distribution in range [0,1]. Where, $size\ of\ \boldsymbol{v} = size\ of\ the\ nodes\ in\ the\ layer$
- $\forall v_i \in \boldsymbol{v} : v_i \leftarrow 0\ if\ v_i < p\ else\ 1.$ i.e. we will take each values in $\boldsymbol{v}$ and if those values are less than p we set it to 0
- Final activation $\widetilde{\boldsymbol{a}} = \boldsymbol{a} \odot \boldsymbol{v}$ [$\odot$ denotes element wise product]
- By this $p \times 100\%$ of the activations $\boldsymbol{a}$ will be zeroed and rest $(1 - p) \times 100\%$ of the activation will be active.
- The surviving activations are scaled by $\frac{1}{1-p}$ to keep the expected value of the activations consistent.

During inference (making predictions):

- No dropout is applied.
- All neurons are active.
- No scaling is needed, since scaling was already handled during training.

Dropout



Originally, drop probability 0.5

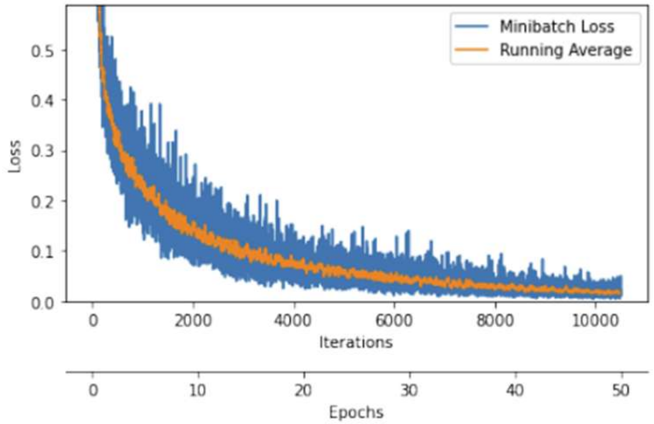
(but 0.2-0.8 also common now)

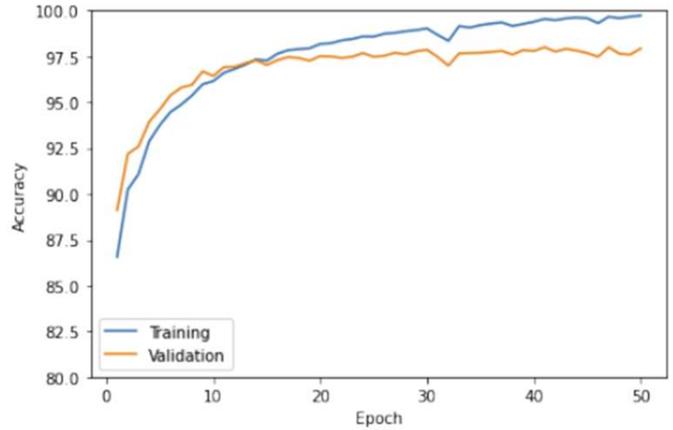
- Dropout reduces overfitting as it prevents the network from relying too heavily on specific neurons or features.
- It acts like an ensemble. Each training step (epoch) samples a different “thinned” subnetwork, and at the inference, you use the average effect of many subnetworks.
- It improves generalization and leads to better performance on unseen/test data.

```python
import torch
import torch.nn as nn

class MLP(nn.Module):

    def __init__(self, input_size, hidden_size_1, hidden_size_2,
                 num_classes, drop_proba):
        super(MLP, self).__init__()

        self.network = nn.Sequential(
            nn.Linear(input_size, hidden_size_1),
            nn.ReLU(),
            nn.Dropout(drop_proba)
            nn.Linear(hidden_size_1, hidden_size_2),
            nn.ReLU(),
            nn.Dropout(drop_proba)
            nn.Linear(hidden_size_2, num_classes)
        )

    def forward(self, x):
        logits = self.network(x)
        return logits

for epoch in range(NUM_EPOCHS):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):
        ### codes for training ###

    model.eval()
    with torch.no_grad():
        ### codes for eval ###
```

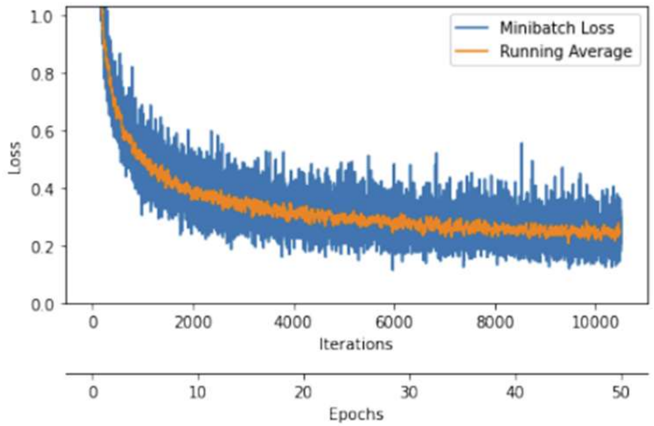
Dropout

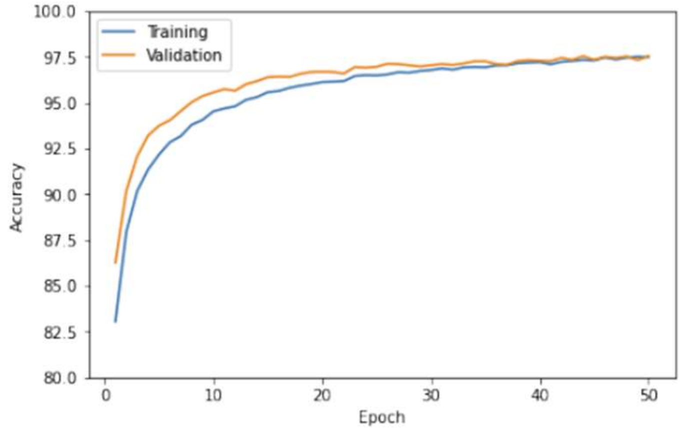
Without dropout:





With 50% dropout:

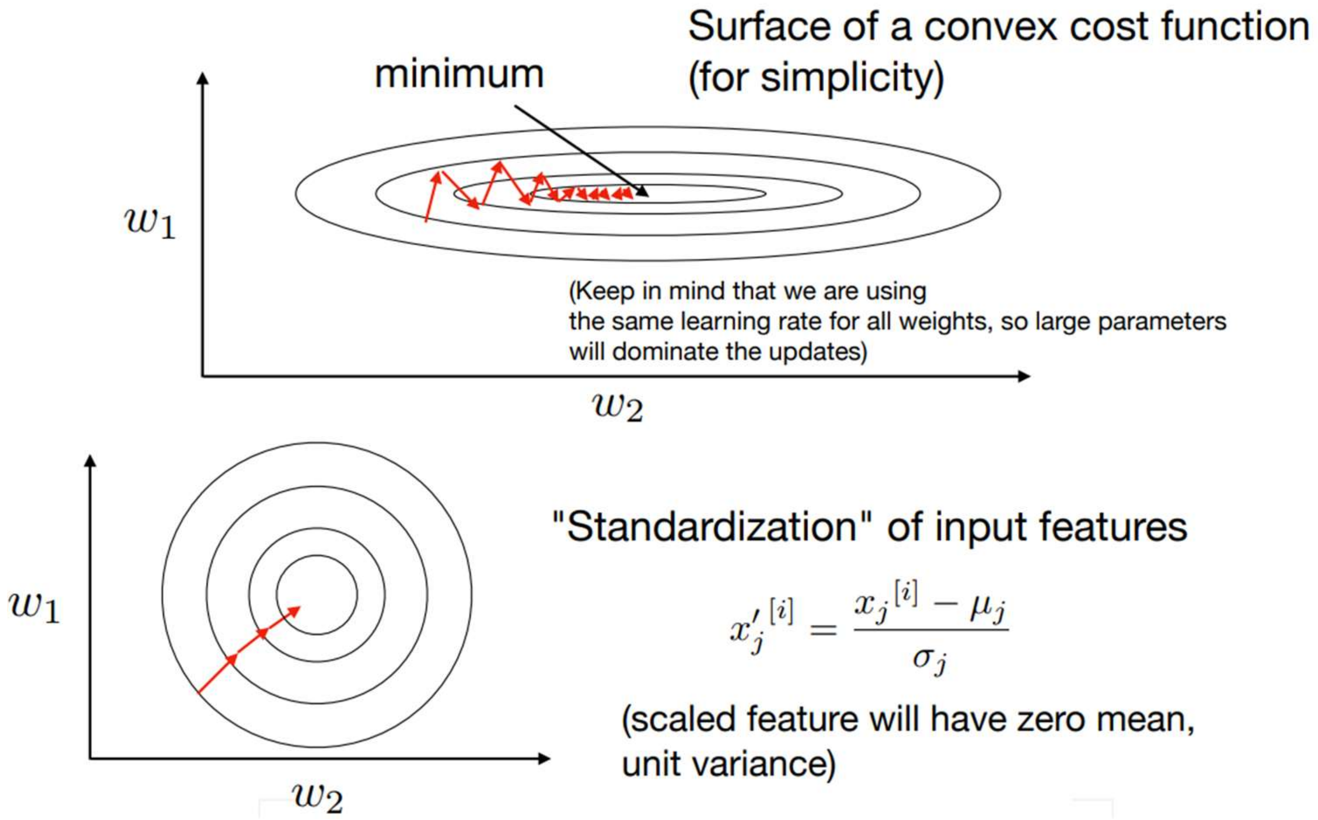




Section-2: Normalization

Feature/Input Normalization

Why do we normalize inputs for Gradient Descent?

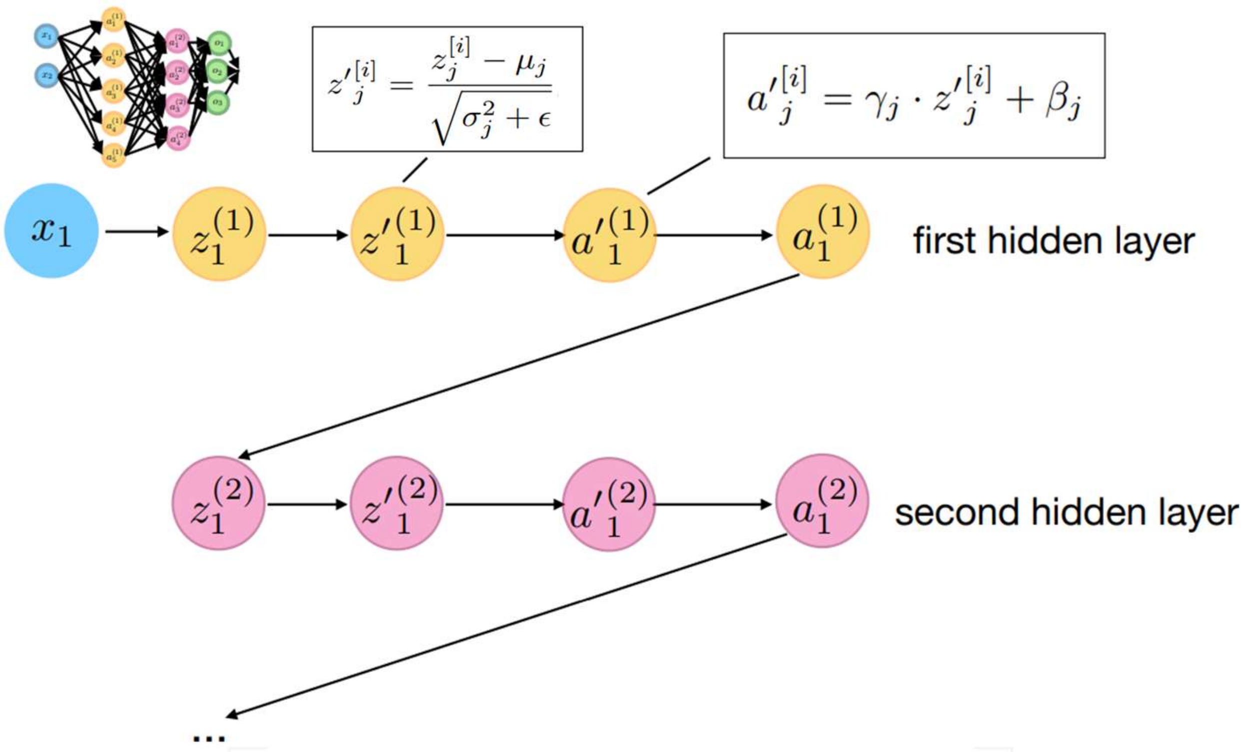


However, normalizing the inputs / features only affects the first hidden layer. What about the other hidden/output layers?

Batch Normalization

Key Concepts

- Normalizes the hidden layer inputs.
- Helps with exploding /vanishing gradient problems.
- Can increase raining stability and convergence rate.
- Can be understood as additional (normalization) layers (with additional parameters).



γ_j and β_j are the learning parameters

β_j makes the bias redundant

Hence, we use batchnorm without bias

Batch Normalization with Dropout

```
import torch
import torch.nn as nn

class MLP(torch.nn.Module):

    def __init__(self, input_size, hidden_size_1, hidden_size_2,
                 num_classes, drop_proba):
        super(MLP, self).__init__()

        self.network = nn.Sequential(
            nn.Linear(input_size, hidden_size_1, bias=False),
            nn.BatchNorm1d(hidden_size_1),
            nn.ReLU(),
            nn.Dropout(drop_proba),
            nn.Linear(hidden_size_1, hidden_size_2, bias=False),
            nn.BatchNorm1d(hidden_size_2),
            nn.ReLU(),
            nn.Dropout(drop_proba),
            nn.Linear(hidden_size_2, num_classes)
        )

    def forward(self, x):
        logits = self.network(x)
        return logits

for epoch in range(NUM_EPOCHS):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):
        ### codes for traning ###

    model.eval()
    with torch.no_grad():
        ### codes for eval ###
```

`Linear → BatchNorm1d → ReLU → Dropout`

During inference/prediction BatchNorm use global training set mean and varaince

Uses exponentially weighted average (moving average) of mean and variance.

mean = momentum * mean + (1-momentum) * sample_mean

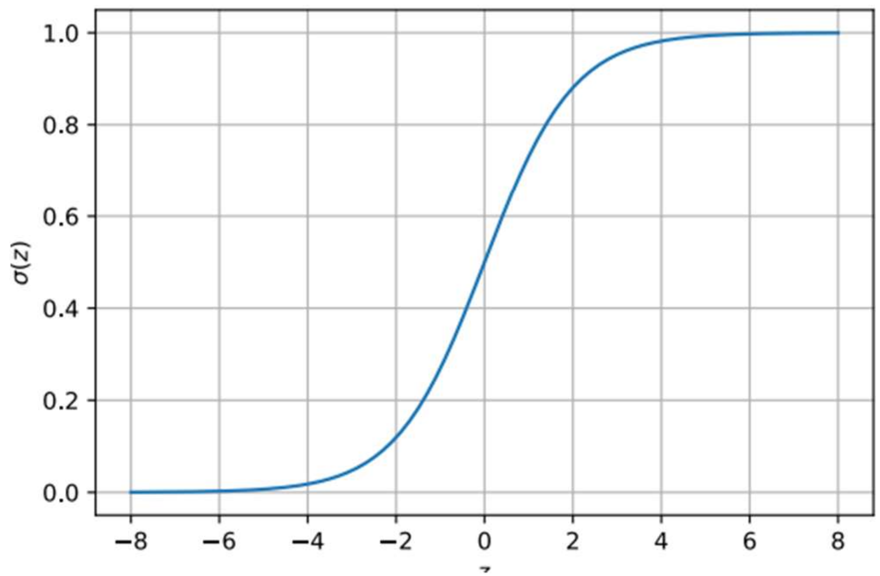
Batch Normalization

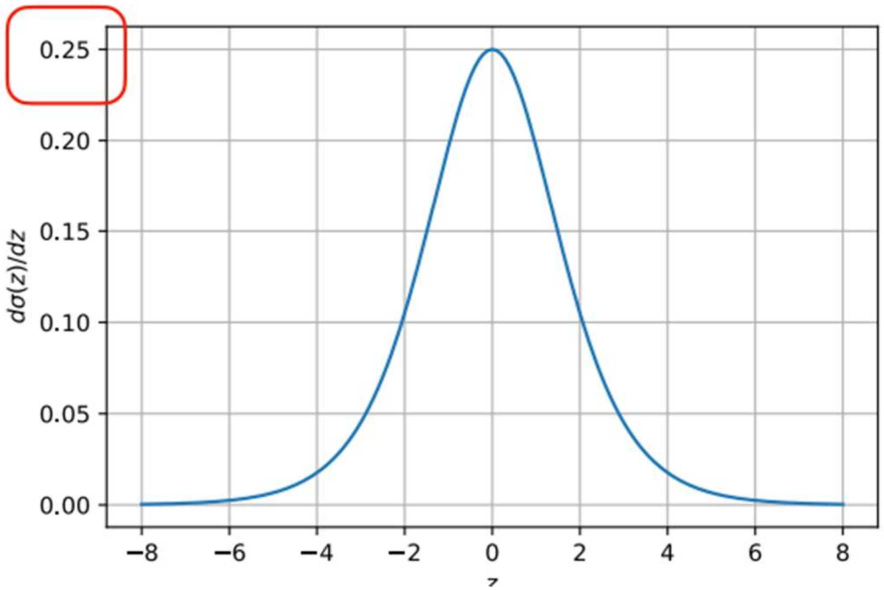
```python
def train_model(model, num_epochs, train_loader,
                valid_loader, test_loader, optimizer, device):

    start_time = time.time()
    minibatch_loss_list, train_acc_list, valid_acc_list = [], [], []
    for epoch in range(num_epochs):

        model.train()
        for batch_idx, (features, targets) in enumerate(train_loader):

            features = features.to(device)
            targets = targets.to(device)

            # ## FORWARD AND BACK PROP
            logits = model(features)
            loss = torch.nn.functional.cross_entropy(logits, targets)
            optimizer.zero_grad()

            loss.backward()

            # ## UPDATE MODEL PARAMETERS
            optimizer.step()

            # ## LOGGING
            minibatch_loss_list.append(loss.item())
            if not batch_idx % 50:
                print(f'Epoch: {epoch+1:03d}/{num_epochs:03d} '
                      f'| Batch {batch_idx:04d}/{len(train_loader):04d} '
                      f'| Loss: {loss:.4f}')

        model.eval()
        with torch.no_grad():  # save memory during inference
            train_acc = compute_accuracy(model, train_loader, device=device)
```

don't forget `model.train()`
and `model.eval()`
in training and test loops

Vanishing and Exploding Gradient Problem

$$\sigma(z) = \frac{1}{1+e^{-z}} \qquad \frac{d}{dz}\sigma(z) = \frac{e^{-z}}{(1+e^{-z})^2} = \sigma(z)(1-\sigma(z))$$





Assume, we have the largest gradient:

$$\frac{d}{dz}\sigma(0.0) = \sigma(0.0)(1-\sigma(0.0)) = 0.25$$

Even then, for, e.g., 10 layers, we degrade the other gradients substantially!

$$0.25^{10} \approx 10^{-6}$$

Section-3: Weight initialization

Why weight initialization matters?

- Neural networks learn by adjusting weights.
- Poor initialization can cause:
  - Vanishing gradients: happens when weights are too small.
  - Exploding gradients: happens when weights are too large.
  - Slow convergence: the weight update happens slowly.
  - Getting stuck in symmetry: if all weight start with the same value, neurons in a layer receive the same input and produces identical outputs. Backprop gives them identical gradient and they keep updating identically and fails to learn the diversity in features.
- Good initialization gives:
  - Stable gradients: keeps the signal flow stable by avoiding exploding / vanishing gradient problem
  - Faster training and faster convergence.
  - Better accuracy as the neurons in each layer will learn diverse aspects of the features.

Vanishing / Exploding gradient caused by too small or too large weights

Backpropagation repeatedly applies matrix multiplications by weight matrices *and* multiplies by activation derivatives. If the typical scale of those multipliers is $\ll 1$, the product decays exponentially with depth → **vanishing gradients**. If the scale is $\gg 1$, the product can blow up → **exploding gradients**.

Glorot and He initialization

Glorot (Xavier) initialization [Glorot & Bengio (2010)]

Glorot initialization, also known as **Xavier initialization**, is designed for activation functions that are symmetric around zero, like the **tanh** function or the **sigmoid** function. The method calculates the initial weights from a random distribution with a specific variance.

Uniform Distribution

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, +\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}\right]$$

Normal Distribution

$$W \sim N\left(0, \sqrt{\frac{2}{n_{in} + n_{out}}}\right)$$

Where n_{in} is the number of input units (fan-in) to the layer and n_{out} is the number of output units (fan-out) from the layer

He initialization [Kaiming He et al. (2015)]

He initialization is a variation of Glorot initialization specifically designed for **ReLU (Rectified Linear Unit)** and its variants (like Leaky ReLU). Because ReLU functions are not symmetric and output zero for all negative inputs, they can cause the variance to drop by half at each layer. He initialization accounts for this by adjusting the variance of the initial weights.

Uniform Distribution

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_{in}}}, +\frac{\sqrt{6}}{\sqrt{n_{in}}}\right]$$

Normal Distribution

$$W \sim N\left(0, \sqrt{\frac{2}{n_{in}}}\right)$$

Weight initialization in PyTorch

PyTorch by default initializes weights by a variation of **Kaiming Uniform** (He initialization)

$$W \sim U\left[-\sqrt{\frac{1}{n_{in}}}, +\sqrt{\frac{1}{n_{in}}}\right] \; and\ biases\ are\ initilizaed\ to\ \mathbf{0}\ if\ bias = True$$

PyTorch uses a different version of Kaiming Uniform initialization because this initialization method generalizes well for different layers (like linear, conv2d etc.) and different activation functions. However, one can change the default initialization by using **`torch.nn.init`** module.

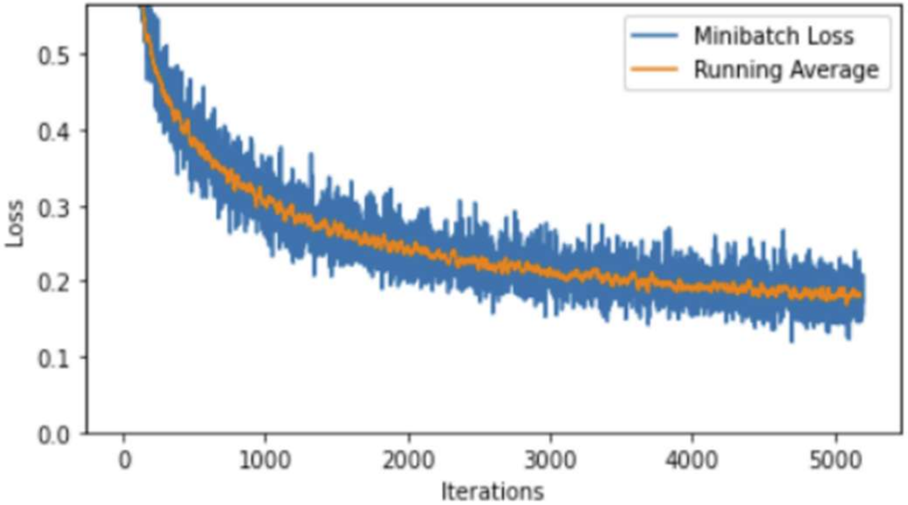
```
import torch
import torch.nn as nn

class MLP(torch.nn.Module):

    def __init__(self, num_features, num_hidden, num_classes):
        super().__init__()

        ### 1st hidden layer
        self.linear_1 = torch.nn.Linear(num_features, num_hidden)
        nn.init.kaiming_uniform_(self.linear_1.weight, mode='fan_in', nonlinearity='relu') # initializing the weights
        nn.init.zeros_(self.linear_1.bias)                                                 # initializing the bias

        ### Output layer
        self.linear_out = torch.nn.Linear(num_hidden, num_classes)
        nn.init.normal_(self.linear_out.weight, mean=0.0, std=0.1)   # initializing the weights
        nn.init.zeros_(self.linear_out.bias)                         # initializing the bias

    def forward(self, x):
        ### codes for forward pass ###
```

Section-4: Learning Rate Decay

Learning rate decay

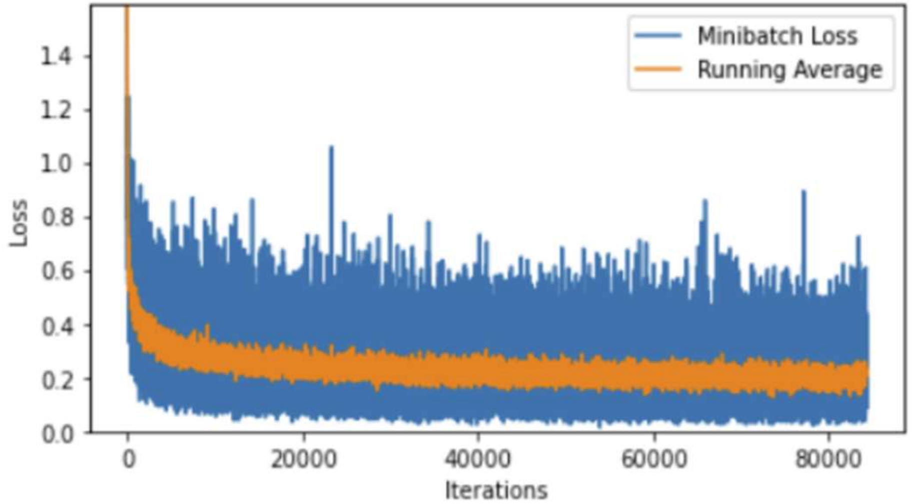
batchsize-1024

```
Epoch: 100/100 | Train: 98.45% | Validation: 97.67%
Time elapsed: 4.38 min
Total Training Time: 4.38 min
Test accuracy 97.08%
```



batchsize-64

```
Epoch: 100/100 | Train: 98.50% | Validation: 97.65%
Time elapsed: 5.59 min
Total Training Time: 5.59 min
Test accuracy 97.18%
```



- Batch effects: minibatches are samples of training set, hence minibatch loss and gradients are approximations.
- Hence, we usually get oscillations.
- To dampen the oscillations towards the end of the training, we can decay the learning rate.

Learning rate decay

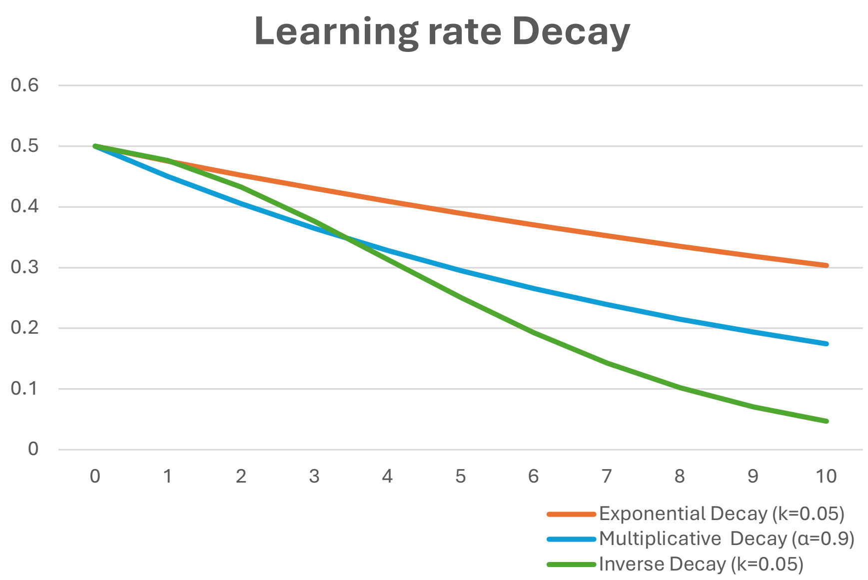
Most common variants of learning rate decay

1. Exponential Decay: $\eta_t := \eta_0 \cdot e^{-kt}$; where k is the decay rate.

2. Multiplying the decay with some positive fraction: $\eta_t := \alpha \cdot \eta_{t-1}$; where $0 < \alpha < 1$.

3. Inverse decay: $\eta_t := \frac{\eta_0}{1+k \cdot t}$; where k is the decay parameter.

And there are many more…



Caveats of using learning rate decay

- Decay can happen too early if the decay parameter is not tuned properly.
- Practical tip: try to train the model without learning rate decay first, then add it later.
- You can also use validation performance (e.g. validation accuracy) to judge whether LR-decay is useful.

Learning rate schedulers in PyTorch

Option-1: Call your own function at the end of each epoch

```
def adjust_learning_rate(optimizer, epoch, initial_lr, decay_rate):
    """Exponential decay every 10 epochs"""
    if not epoch % 10:
        lr = initial_lr * torch.exp(-decay_rate*epoch)
        for param_group in optimizer.param_groups:
            param_group['lr'] = lr
```

Option-2: Use one of the built in tools in **`torch.optim.lr_scheduler`** module

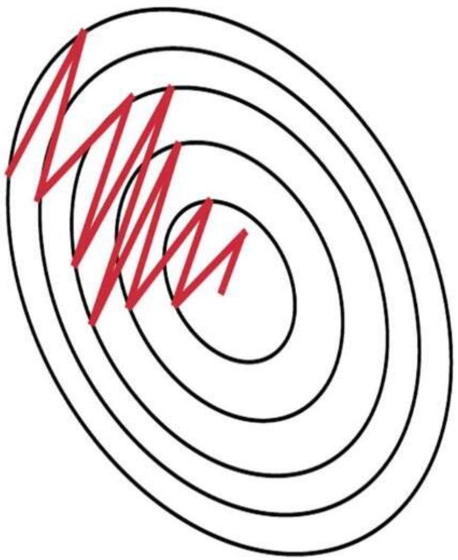
```
# Defining an optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=0.05)
# Defining Learning rate scheduler
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.1)

for epoch in range(5):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):
        features = features.to(DEVICE)
        targets = targets.to(DEVICE)
        logits, probas = model(features)
        cost = F.cross_entropy(logits, targets)
        optimizer.zero_grad()
        cost.backward()
        optimizer.step() # UPDATE MODEL PARAMETERS

    # UPDATE LEARNING RATE
    scheduler.step() # don't have to do it every epoch!
```

Section-5: Momentum and Adaptive Learning Rates

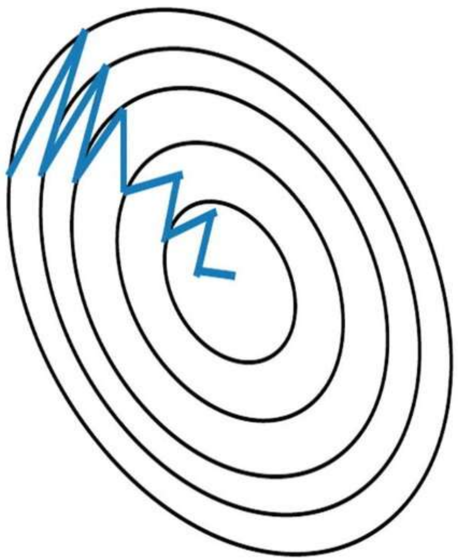
Training with momentum

Key Concept: In momentum learning, we try to accelerate convergence by dampening oscillations using "velocity" (the speed of the "movement" from previous updates)

Momentum helps with dampening oscillations.



Stochastic Gradient Descent **withhout** Momentum



Stochastic Gradient Descent **with** Momentum

Training with momentum

Often referred to as "velocity" v

"velocity" from the previous iteration

$$\Delta w_{i,j}(t) := \alpha \cdot \Delta w_{i,j}(t-1) + \eta \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}}(t)$$

Usually, we choose a momentum rate between 0.9 and 0.999; you can think of it as a "friction" or "dampening" parameter

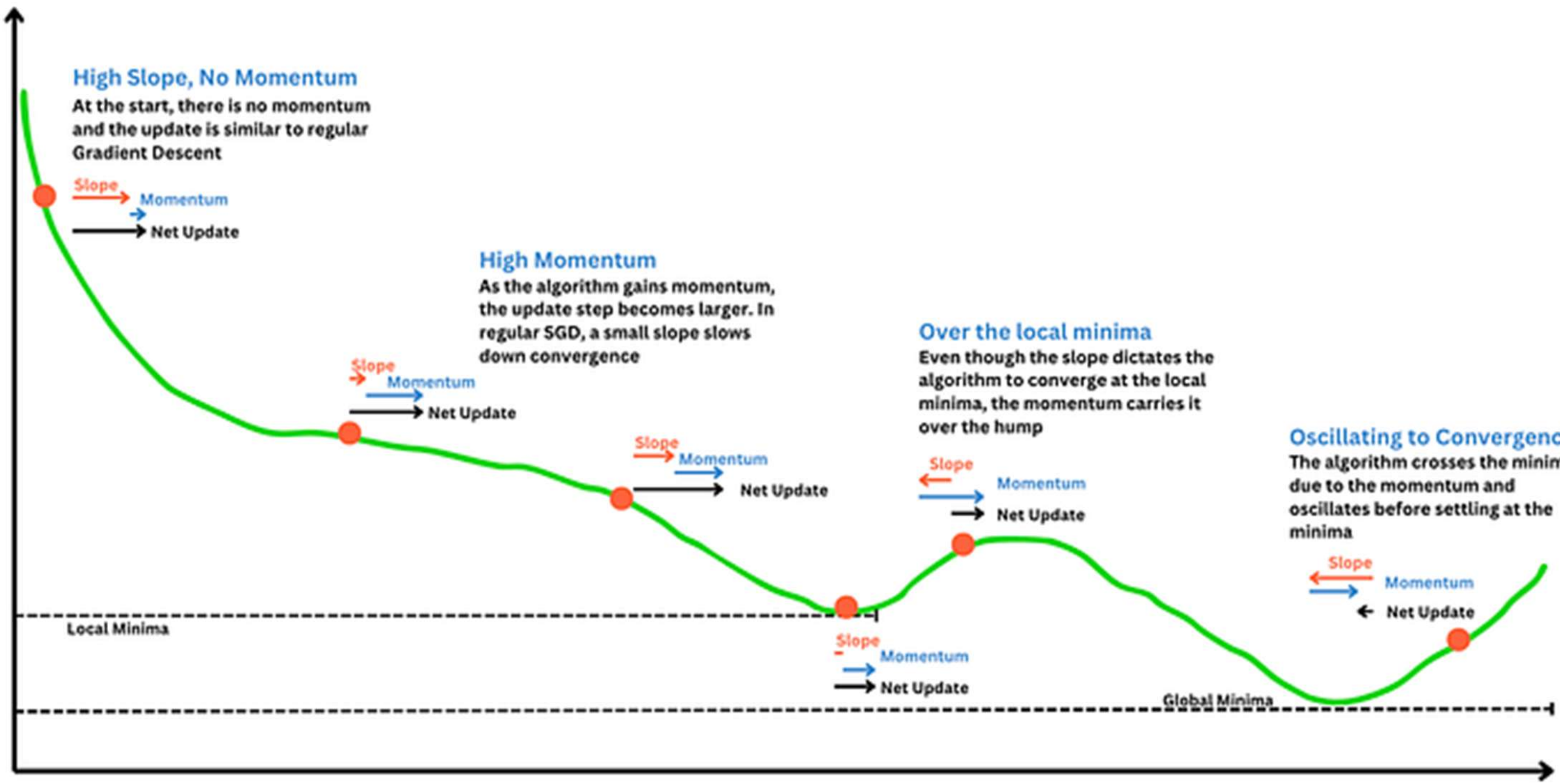
Regular partial derivative/ gradient multiplied by learning rate at current time step t

Weight update using the velocity vector:

$$w_{i,j}(t+1) := w_{i,j}(t) - \Delta w_{i,j}(t)$$

Training with momentum

Momentum helps with escaping the local minima traps



Nesterov’s Accelerated Gradient

Nesterov's Accelerated Gradient (NAG), also known as Nesterov's momentum, is a variation of the standard momentum method. It's a "look-ahead" technique that often leads to faster and more stable convergence.

The key difference is that instead of calculating the gradient at the current position, NAG computes the gradient at a point where the parameters *will be* after applying the current velocity. This allows the optimizer to "see ahead" and make a more informed correction.

Normal momentum equation:

$$v_t = \alpha\, v_{t-1} + \eta\, \nabla_w \mathcal{L}(w_t)$$

$$w_{t+1} = w_t - v_t$$

Nesterov’s momentum (NAG) equation:

$$v_t = \alpha\, v_{t-1} + \eta\, \nabla_w \mathcal{L}(w_t - \alpha\, v_{t-1})$$

$$w_{t+1} = w_t - v_t$$

The term $(w_t - \alpha\, v_{t-1})$ represents the "look-ahead" position. By calculating the gradient at this projected position, NAG can anticipate the change in direction and slow down earlier, which helps to avoid overshooting a minimum.

Momentum and NAG in PyTorch

```
# SGD with normal momentum
torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.99)

# SGD with Nesterov momentum
torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.99, nesterov=True)
```

Adaptive Learning Rates

Key Concepts:

- Decrease learning if the gradient changes its direction.
- Increase learning if the gradient stays constant.

Here are some of the most common adaptive learning rate algorithms:

- **AdaGrad** (Adaptive Gradient Algorithm): Best for sparse data problems, such as in NLP or recommendation systems.
- **RMSprop** (Root Mean Square Propagation): Often used in deep neural networks (especially in RNNs)
- **ADAM** (Adaptive Momentum Estimation): Used in wide range of deep learning tasks.
- **NADAM (**Nesterov-accelerated Adaptive Moment Estimation): Potentially offers faster convergence on complex models.
- **ADAMW** (ADAM with Weight Decay): Ideal for training large-scale models like Transformers.
- **AdaDelta**: Useful when a learning rate is not pre-defined, as it dynamically adjusts the step size.

We will talk about ADAM optimization algorithm in more details. ADAM is very popular and widely used in different times of neural network.

Adaptive Moment Estimation (ADAM)

ADAM Algorithm Initialization

- w_0 : $Initial\ parameter\ vector$
- $m_0 \leftarrow 0\ (Initialize\ first\ moment\ vector)$
- $v_0 \leftarrow 0\ (Initialize\ second\ moment\ vector)$

ADAM Algorithm Weight Update

- $g_t \leftarrow \nabla_w \mathcal{L}(w_t)$: $Get\ the\ gradients\ of\ loss\ function\ \mathcal{L}\ at\ timestep\ 't'$
- $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$: $Update\ biased\ first\ moment\ estimate$
- $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$: $Compute\ bias - corrected\ first\ moment\ estimate$
- $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$: $Update\ biased\ second\ raw\ moment\ estimate$
- $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$: $Compute\ bias - corrected\ second\ raw\ moment\ estimate$
- $w_{t+1} \leftarrow w_t - \gamma \cdot \frac{\hat{m}_t}{(\sqrt{\hat{v}_t} + \epsilon)}$: $Update\ parameters\ (where\ \gamma\ is\ the\ learning\ rate)$

Here $g_t^2 = g_t \odot g_t$: $Element\ wise\ product\ of\ the\ gradient$

Good default settings that can be used for most of the cases:

- $\gamma\ (learning\ rate)$: 0.001
- β_1 : 0.9
- β_2 : 0.999
- ϵ : 10^{-8}

Adaptive Moment Estimation (ADAM)

Why bias correction is needed in ADAM Algorithm

Since m_0 is initialized to 0, in the first few iterations, m_t will be biased towards zero, especially if β_1 is close to 1. To correct this, the algorithm divides m_t by a term that accounts for the number of steps. The **bias-corrected first moment** ($\widehat{m}_t$), is:

$$\widehat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$$

Like the first moment, v_0 is initialized to 0, causing a similar bias. Since β_2 is often very close to 1, this bias can be even more pronounced in the early stages, leading to an artificially small denominator in the final update rule and potentially causing the initial steps to be too large. The **bias-corrected second moment** ($\hat{v}_t$), is:

$$\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$$

ADAM optimizer in PyTorch

```
# Adam optimizer with deafult settings
torch.optim.Adam(model.parameters(), lr=0.001, betas=(0.9, 0.999), eps=1e-8)
```

Section-6: Hyperparameters in Neural Network

Hyperparameters in Neural Network

Hyperparameters are user-defined and unlike parameters (weights and biases) they are not learnt by data.

Following are the list of different types of hyperparameters mainly in MLP, though few of them are common to other type of neural network architecture.
The grayed out hyperparameters are usually not tuned during hyperparameter tuning.

Architectural Hyperparameters

- Number of hidden layers
- Number of hidden units in each hidden layers
- Regularization technique (for dropout the dropout probability is a hyperparameter)
- Activation functions in hidden layers *(ReLU is often default choice)*

Learning based Hyperparameters

- Number of epochs
- Minibatch size
- Learning rates *(often scheduler is used)*
- Optimization algorithm (often the default parameters of a particular optimization algorithm like ADAM works good in practice)

The hyperparameter optimization can be done in various ways like: Random Search, Grid Search or Bayesian optimization. However, in most of the cases the intuition are used to train the neural network efficiently. For example: if the loss vs epoch curve shows zig-zag or oscillatory pattern then it’s a good idea to increase the mini-batch size.

Thank You