Data-intensive space engineering

Lecture 8

Carlos Sanmiguel Vila

Improving NNs Training

In this lecture, we review a few techniques to improve the performance of (deep) neural networks

- **Vanishing/exploding gradients**
  - Gradients grow smaller and smaller or larger and larger as we flow backwards through the network during training
- **Regularization**
  - We might not have enough training data for training a large network or data instances are too noisy
  - Batch normalization and dropout
  - L_1 and L_2 regularization
- **Optimizers**
  - Various optimization methods can speed up training large neural networks

Vanishing/exploding gradients problem

- When gradients get smaller and smaller, the Gradient Descent update leaves many connection weights unchanged
  - **Known as the vanishing gradients problem**

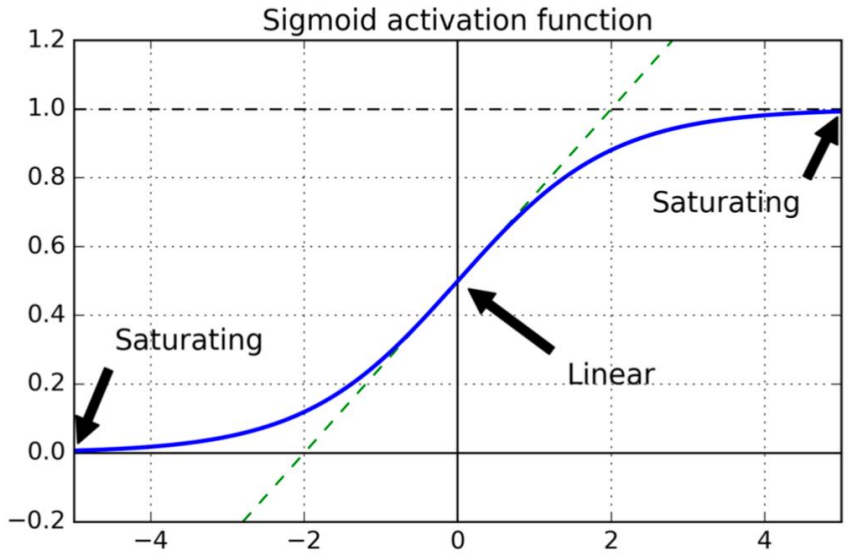
- When gradients grow bigger and bigger, some layers get large weight updates and the algorithm diverges
  - **Known as the exploding gradients problem**

- One of the main reasons deep neural networks were abandoned in 2000s

- It appears there are two main factors
  - Weight initialization
  - Sigmoid activation function

Activation function saturation

- When inputs become large (negative or positive), the function saturates at 0 or 1
- Derivative extremely close to 0
- No gradient to propagate back through the network



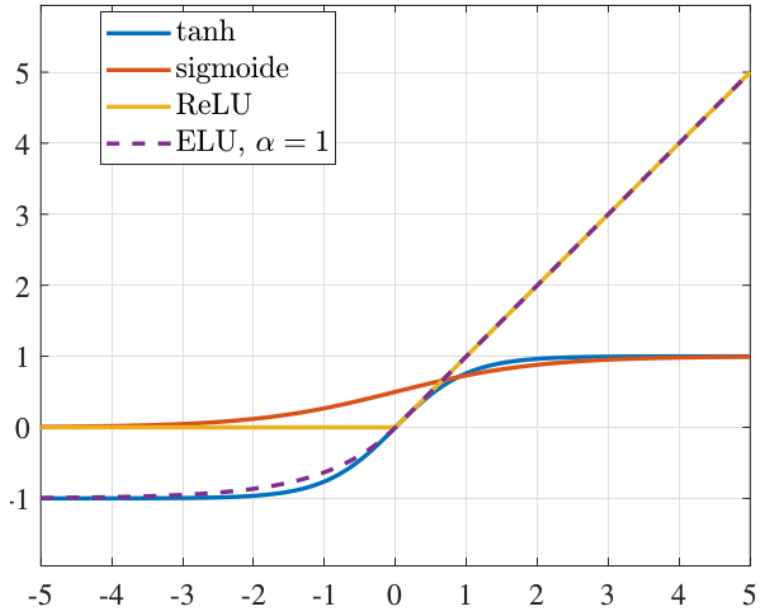
Why Does Weight Initialization Matter?

The main issues with poor weight initialization are:

- **Vanishing gradients:** If weights are initialized too small, they can become exponentially smaller during backpropagation as they move backwards through the layers. This causes the gradients to “vanish,” resulting in extremely slow learning or even complete stagnation.
- **Exploding gradients:** If weights are initialized too large, the gradients can become very large during backpropagation, which may cause the model’s weights to grow uncontrollably, leading to divergence in training**.**
- **Symmetry breaking:** Initializing all weights to the same value (e.g., 0) would cause neurons to learn the same features, effectively making the neurons redundant. Random initialization is used to prevent this.

Nonsaturating activation functions

- We know that ReLU activation function does not saturate for positive values, and it is fast to compute
- However, ReLU is not perfect!
- Dying ReLU problem: some neurons effectively "die," meaning they stop producing anything other than 0
- Solution: ELU(x)=x if x>0, else $\alpha(e^x-1)$ with $0 < \alpha$
- If $\alpha = 1$, the function is smooth everywhere including $x = 0$



Glorot and He Initialization

- **Goal**: the variance of the outputs of each layer to be equal to the variance of its inputs
- Number of inputs: fan_in
- Number of output neurons: fan_out
- Let us define fan_avg = (fan_in + fan_out)/2

Equation 11-1. Glorot initialization (when using the logistic activation function)

Normal distribution with mean 0 and variance $\sigma^2 = \frac{1}{fan_{avg}}$

Or a uniform distribution between $-r$ and $+r$, with $r = \sqrt{\frac{3}{fan_{avg}}}$

Table 11-1. Initialization parameters for each type of activation function

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, tanh, logistic, softmax	$1 / fan_{avg}$
He	ReLU and variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

Glorot and He Initialization

- **Xavier (Glorot) Initialization:** Designed for activation functions like sigmoid and tanh, where it attempts to keep the variance of the inputs and outputs consistent across layers. The weights are drawn from a distribution with a variance based on the number of input and output units.

- **He Initialization (Kaiming Initialization)**: Developed for ReLU-like activation functions, which "kill" negative activations by setting them to zero. He initialization accounts for the asymmetry of ReLU. It sets the weights to a distribution that helps maintain a good flow of gradients during training.

- By default, PyTorch uses a form of Xavier (Glorot) uniform initialization for the weights in layers like nn.Linear and nn.Conv2d, with biases initialized to zeros.

Glorot and He Initialization

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

# Manually initialize weights using Xavier uniform
def init_weights(m):
    if isinstance(m, nn.Linear):
        init.xavier_uniform_(m.weight)
        init.zeros_(m.bias)

model = SimpleModel()
model.apply(init_weights)  # Apply custom initialization
```

python Copiar código

```python
def init_weights(m):
    if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
        init.kaiming_uniform_(m.weight, nonlinearity='relu')  # He initialization
        if m.bias is not None:
            init.zeros_(m.bias)  # Set biases to zero

model.apply(init_weights)
```

Choosing the Right Initialization

- Xavier Initialization is typically used for sigmoid and tanh activation functions, where the gradients can vanish.
- He Initialization is preferred for ReLU and its variants, as it helps maintain gradient flow by initializing weights with larger values.
- ELU can also benefit from He Initialization, but some experimentation might be necessary.
- If you're unsure, PyTorch’s default initialization (Xavier) is generally a good starting point for most networks, and you can experiment with custom initialization if necessary.

Batch normalization

- Batch normalization is a regularization technique, although it primarily aims to stabilize and speed up training. It works by normalizing the outputs of each layer to have a mean of 0 and a variance of 1 across the batch. This reduces internal covariate shift, making the model's convergence easier.
- For each batch, batch normalization normalizes the output h of a layer:

$$\hat{h} = \frac{h - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

where

- μ_B is the mean of the batch,
- σ_B^2 is the variance of the batch,
- ϵ is a small constant for numerical stability.
- Batch normalization also introduces two trainable parameters γ and β, which allow the network to learn the optimal scale and shift for the normalized outputs.

Equation 11-3. Batch Normalization algorithm

1. $\mathbf{\mu}_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$
2. $\mathbf{\sigma}_B^{\,2} = \frac{1}{m_B} \sum_{i=1}^{m_B} \left(\mathbf{x}^{(i)} - \mathbf{\mu}_B\right)^2$
3. $\widehat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mathbf{\mu}_B}{\sqrt{\mathbf{\sigma}_B^{\,2} + \varepsilon}}$
4. $\mathbf{z}^{(i)} = \mathbf{\gamma} \otimes \widehat{\mathbf{x}}^{(i)} + \mathbf{\beta}$

Batch normalization

```python
class MLPWithBatchNorm(nn.Module):
    def __init__(self):
        super(MLPWithBatchNorm, self).__init__()
        self.fc1 = nn.Linear(784, 512)
        self.bn1 = nn.BatchNorm1d(512)  # BatchNorm after first fully connected layer
        self.fc2 = nn.Linear(512, 256)
        self.bn2 = nn.BatchNorm1d(256)  # BatchNorm after second fully connected layer
        self.fc3 = nn.Linear(256, 10)

    def forward(self, x):
        x = F.relu(self.bn1(self.fc1(x)))  # Apply batch norm followed by ReLU
        x = F.relu(self.bn2(self.fc2(x)))  # Apply batch norm followed by ReLU
        x = self.fc3(x)
        return x

# Example of usage
model = MLPWithBatchNorm()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Batch normalization

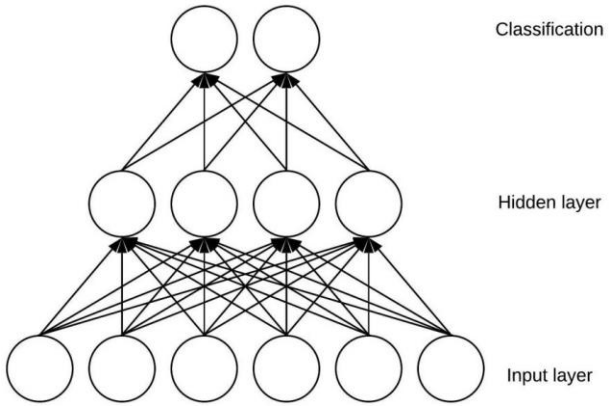
In this model, we apply batch normalization after the linear transformations and before the activation function (ReLU). This ensures that the inputs to the next layer have stable distributions, making the model easier to train.

Effects of Batch Normalization:

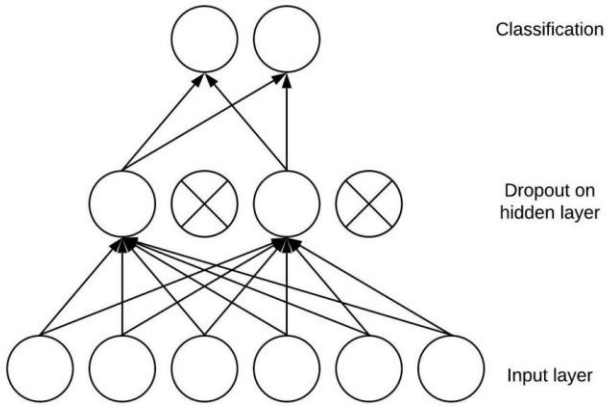
- **Faster Convergence**: By normalizing the outputs, the network can converge faster during training
- **Smoothing the Loss Landscape**: Batch normalization reduces the problem of vanishing/exploding gradients
- **Regularization Effect**: Batch normalization has a mild regularization effect, as each mini-batch's statistics are slightly different

Dropout

- Dropout is a simple but effective regularization technique. It works by randomly "dropping" or setting a fraction of neurons to zero during training, which prevents the network from becoming too reliant on specific neurons and encourages it to learn more robust, distributed representations
- If a layer has a set of neurons h={h1,h2,…,hn},during training, Dropout randomly sets some neurons hi to zero with a probability p. The remaining neurons are scaled by $\frac{1}{1-p}$ to maintain the overall contribution to the next layer.



Without Dropout



With Dropout

Dropout

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

class MLPWithDropout(nn.Module):
    def __init__(self):
        super(MLPWithDropout, self).__init__()
        self.fc1 = nn.Linear(784, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 10)
        self.dropout = nn.Dropout(p=0.5)  # Dropout with probability 0.5

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout(x)  # Apply dropout after first layer
        x = F.relu(self.fc2(x))
        x = self.dropout(x)  # Apply dropout after second layer
        x = self.fc3(x)
        return x

# Example of usage
model = MLPWithDropout()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Dropout

Dropout helps reduce overfitting by forcing the network to learn more robust features. It can slow down convergence since the network is effectively training different subnetworks each time, but the final model tends to generalize better.

Combining Dropout and Batch Normalization:

- You can combine both techniques in a network, but typically **dropout** is applied after fully connected layers, while **batch normalization** is applied between the linear layer and the activation function.

Key Takeaways:

- **Dropout**: Prevents overfitting by randomly dropping neurons during training. It is typically used after fully connected layers in both MLPs and CNNs.
- **Batch Normalization**: Speeds up training by normalizing the output of each layer to reduce internal covariate shift. It can be used after linear layers (or convolutional layers for CNNs) and before activation functions.

Dropout

```python
class MLPWithDropoutAndBatchNorm(nn.Module):
    def __init__(self):
        super(MLPWithDropoutAndBatchNorm, self).__init__()
        self.fc1 = nn.Linear(784, 512)
        self.bn1 = nn.BatchNorm1d(512)
        self.fc2 = nn.Linear(512, 256)
        self.bn2 = nn.BatchNorm1d(256)
        self.fc3 = nn.Linear(256, 10)
        self.dropout = nn.Dropout(p=0.5)

    def forward(self, x):
        x = F.relu(self.bn1(self.fc1(x)))
        x = self.dropout(x)
        x = F.relu(self.bn2(self.fc2(x)))
        x = self.dropout(x)
        x = self.fc3(x)
        return x

# Example of usage
model = MLPWithDropoutAndBatchNorm()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

L1 and L2 Regularization

L1 and L2 regularization are two of the most used techniques for adding constraints to model weights during training to reduce overfitting. They are also known as weight regularization because they penalize the size of the model's weights. By introducing a penalty term to the loss function, these methods discourage the model from learning overly complex patterns that may not generalize well to unseen data.

L2 Regularization (Ridge Regularization)

- L2 regularization adds a penalty proportional to the **square** of the magnitude of the weights to the loss function. This helps prevent the weights from becoming too large, which can help generalize new data.

L1 Regularization (Lasso Regularization)

- L1 regularization adds a penalty proportional to the **absolute value** of the weights to the loss function. This can lead to sparse models where many weights are driven to exactly zero, effectively performing feature selection by removing irrelevant features.

L1 and L2 Regularization

Key Differences Between L1 and L2 Regularization

- L1 Regularization: Drives some weights to exactly zero, leading to a sparse model that can act as feature selection. This is particularly useful in high-dimensional datasets where many features may be irrelevant.
- L2 Regularization: Shrinks weights uniformly but does not necessarily drive them to zero. It helps distribute the impact of different features more smoothly and can lead to more balanced models.

When to Use L1 or L2

- L1: Use L1 regularization when you suspect that some features are irrelevant, and you want the model to automatically select a subset of important features. This is especially useful in high-dimensional feature spaces, like text processing or genomics.
- L2: Use L2 regularization when you want to prevent large weights but don't expect a large number of irrelevant features. It’s useful when all features are expected to contribute to the output but may overfit without regularization.

L1 and L2 Regularization

In PyTorch, L2 regularization is often applied through the weight_decay parameter in optimizers such as Adam, SGD, etc.

```
# Use L2 regularization by setting the weight_decay parameter
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=0.01)  # L2 regularizati
```

L1 regularization

```
# Define a function for L1 regularization
def l1_regularization(model, lambda_l1):
    l1_norm = sum(param.abs().sum() for param in model.parameters())
    return lambda_l1 * l1_norm

# Example training loop with L1 regularization
lambda_l1 = 0.001  # Regularization strength for L1

for epoch in range(epochs):
    for batch in train_loader:
        data, target = batch
        optimizer.zero_grad()
        output = model(data)
        loss = loss_fn(output, target)

        # Add L1 regularization to the loss
        loss += l1_regularization(model, lambda_l1)

        loss.backward()
        optimizer.step()
```

Popular Optimizers in PyTorch

In practice, several optimizers build on top of basic gradient descent to make training more efficient, faster, and stable.

Stochastic Gradient Descent (SGD)
SGD is one of the most straightforward optimization techniques. It updates weights using the gradient of a small batch of data, which can make updates noisy but faster.

$$W = W - \eta \nabla_W L$$

```
import torch.optim as optim

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

Drawback: SGD can oscillate a lot near minima because it uses raw gradients. Therefore, it's often combined with techniques like **momentum**.

Popular Optimizers in PyTorch

SGD with Momentum

Momentum is a technique that helps SGD overcome the problem of oscillation, especially in regions where gradients vary greatly in magnitude. It does this by maintaining an exponentially decaying moving average of past gradients.

$$v_t = \beta v_{t-1} + (1-\beta)\nabla_W L$$
$$W = W - \eta v_t$$

Where β is the momentum coefficient (usually between 0.9 and 0.99).
Momentum helps accelerate convergence, particularly in areas where gradients are noisy or highly varied.

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

Popular Optimizers in PyTorch

RMSProp (Root Mean Square Propagation)

RMSProp adapts the learning rate based on the average of recent squared gradients. This prevents the learning rate from becoming too large in directions where gradients are large.

$$v_t = \beta v_{t-1} + (1-\beta)(\nabla_W L)^2$$
$$W = W - \frac{\eta}{\sqrt{v_t} + \epsilon} \nabla_W L$$

```
optimizer = optim.RMSprop(model.parameters(), lr=0.001)
```

Popular Optimizers in PyTorch

Adam (Adaptive Moment Estimation)

Adam is one of the most widely used optimizers because it combines the best of both worlds: momentum and adaptive learning rates. It typically requires less tuning than SGD. Adam maintains both:

- An exponentially decaying average of past gradients (similar to momentum).
- An exponentially decaying average of past squared gradients, which helps adapt the learning rate for each parameter individually.

$$m_t = \beta_1 m_{t_{-1}} + (1-\beta_1)\nabla_W L;\ v_t = \beta v_{t-1} + (1-\beta)\nabla_W L$$

$$\widehat{m_t} = \frac{m_t}{1-\beta_1};\ \widehat{v_t} = \frac{v_t}{1-\beta_2};\ W = W - \eta \frac{\widehat{m_t}}{\sqrt{\widehat{v_t}} + \epsilon}$$

Where:

- $\widehat{m_t}$ is the first moment estimate (momentum-like term).
- $\widehat{v_t}$ is the second moment estimate (RMSProp-like term).
- ϵ is a small constant to prevent division by zero.

```
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Learning Rate Scheduling

The learning rate is one of the most critical hyperparameters in optimization. A high learning rate can lead to divergence, while a low learning rate can result in slow convergence. Therefore, learning rate scheduling is often used to adjust the learning rate dynamically during training.
Common strategies include:

- **Step Decay:** Reduce the learning rate by a factor after a certain number of epochs.

```
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=30, gamma=0.1)
```

This will reduce the learning rate by a factor of 0.1 every 30 epochs.

- **Reduce on Plateau:** Reduce the learning rate if the validation loss stops improving.

scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=10, factor=0.1)

This reduces the learning rate if the loss plateaus.

- **Cosine Annealing:** The learning rate is adjusted following a cosine function over the training epochs, which can help the optimizer converge more smoothly.

```
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=50)
```