Overfitting and Remedies

Find the perfect model complexity, Early stopping, Regularization, Dropout, Data augmentation, and Domain randomization
SpeakUp

Join by URL: https://web.speakup.info/room/join/80740

Join by key: web.speakup.info → use key 80740

Join by saved session: web.speakup.info

Join by QR:
Outline

• Drawing recap for initialization and normalization
  
• Overfitting and its causes
  
• Overfitting remedies
  • Find the perfect model complexity
  • Early stopping
  • Regularization
  • Dropout
  • Data augmentation
  • Domain randomization
Recap: Parameter and Gradient Values

• Take five minutes to draw

• Example: activations with and without proper initialization and normalization
Classroom Etiquette

• We all want to look *effortlessly smart* in front of our peers.

  • It’s a fool’s errand. I’ve noticed it a bit in the class. Might be due to class makeup

• I’ve built my teaching philosophy around the “gift of failure”

  • You need to give me wrong answers

  • You need to be unafraid of being wrong

  • You need to be ready to fail
Overfitting

When your model learns/memorizes the training data and not some property that is useful for inference. ("I've seen this input before... the answer is X.")
Causes of Overfitting

When your model learns/memorizes the training data and not some property that is useful for inference. ("I’ve seen this input before... the answer is X.")

• The model is too complex
  • Too many parameters
  • Too deep
  • Too wide
  • Too much memory
• Parameters are too large (large parameters lead to steep curves)
• The model was trained for too long
• The dataset was too small
Remedy: Find the Perfect Model Complexity

We could theoretically find the perfect model complexity for each problem.

Easy for simple linear problems

Harder for more complicated relationships

\( Y \rightarrow \text{use NN} \)
Hyperparameter Search/Tuning

• Common methods for “finding” good hyperparameters include
  • Manual adjustments
  • Grid search
  • Random search
  • Bayesian optimization
  • Evolutionary optimization
  • (and others)

• I happen to prefer a simple “Twiddle Search”
# Initial values
hyper_params = {
    "learning_rate": 0.1,
    "batch_size": 64,
    "num_layers": 10,
    "dropout": 0.5,
}

# Hyperparameter update factors
hyper_param_updates = {
    "learning_rate": {
        "up": lambda lr: lr * 10,
        "down": lambda lr: lr / 10,
    },
    "batch_size": {
        "up": lambda bs: bs * 2,
        "down": lambda bs: max(bs // 2, 1),
    },
    "num_layers": {
        "up": lambda nl: nl * 2,
        "down": lambda nl: max(nl // 2, 1),
    },
    "dropout": {
        "up": lambda d: min(d + 0.1, 0.9),
        "down": lambda d: max(d - 0.1, 0.1),
    },
}

# Initial quality
best_metric_value = evaluate(hyper_params)

# Cache of hyperparameter value combinations
cache = (hyper_params.values()): best_metric_value
attempts = 1
while not done(best_metric_value, attempts):
    # Choose a hyperparameter and an update direction
    hyper_param = choice(list(hyper_params.keys()))
    update_direction = choice(['up', 'down'])

    # Update the hyperparameter
    current_value = hyper_params[hyper_param]
    new_value = hyper_param_updates[hyper_param][update_direction](current_value)
    new_hyper_params = {**hyper_params, hyper_param: new_value}

    # Check if the hyperparameter value combination has been evaluated before
    if new_hyper_params.values() in cache:
        continue

    attempts += 1

    # Evaluate the new hyperparameter value combination
    metric_value = evaluate(new_hyper_params)
    cache[new_hyper_params.values()] = metric_value

    if metric_value > best_metric_value:
        best_metric_value = metric_value
        hyper_params = new_hyper_params

print("Best metric value: {} (hyper_params)"")
# Initial values
hyper_params = {
    "learning_rate": 0.1,
    "batch_size": 64,
    "num_layers": 10,
    "dropout": 0.5,
}

# Hyperparameter update factors
hyper_param_updates = {
    "learning_rate": {"up": lambda lr: lr * 10, "down": lambda lr: lr / 10},
    "batch_size": {"up": lambda bs: bs * 2, "down": lambda bs: max(bs // 2, 1)},
    "num_layers": {"up": lambda nl: nl * 2, "down": lambda nl: max(nl // 2, 1)},
    "dropout": {"up": lambda d: min(d + 0.1, 0.9), "down": lambda d: max(d - 0.1, 0.1)},
}
# Initial values
hyper_params = {
    "learning_rate": 0.1,
    "batch_size": 64,
    "num_layers": 10,
    "dropout": 0.5
}

# Hyperparameter update factors
hyper_param_updates = {
    "learning_rate": {
        "up": lambda lr: lr * 10,
        "down": lambda lr: lr / 10
    },
    "batch_size": {
        "up": lambda bs: bs * 2,
        "down": lambda bs: max(bs // 2, 1)
    },
    "num_layers": {
        "up": lambda nl: nl * 2,
        "down": lambda nl: max(nl // 2, 1)
    },
    "dropout": {
        "up": lambda d: min(d + 0.1, 0.9),
        "down": lambda d: max(d - 0.1, 0.1)
    }
}

# Initial quality
best_metric_value = evaluate(hyper_params)

# Cache of hyperparameter value combinations
cache = {hyper_params.values(): best_metric_value}

attempts = 1
while not done(best_metric_value, attempts):
    # Choose a hyperparameter and an update direction
    hyper_param = choice(list(hyper_params.keys()))
    update_direction = choice(['up', 'down'])

    # Update the hyperparameter
    current_value = hyper_params[hyper_param]
    new_value = hyper_param_updates[hyper_param][update_direction](current_value)
    new_hyper_params = {
        **hyper_params, hyper_param: new_value
    }

    # Check if the hyperparameter value combination has been evaluated before
    if new_hyper_params.values() in cache:
        continue
    attempts += 1

    # Evaluate the new hyperparameter value combination
    metric_value = evaluate(new_hyper_params)
    cache[new_hyper_params.values()] = metric_value

    if metric_value > best_metric_value:
        best_metric_value = metric_value
        hyper_params = new_hyper_params

    print(f"Best metric value: {best_metric_value}: {hyper_params}")
```python
# Initial values
hyper_params = {
    "learning_rate": 0.1,
    "batch_size": 64,
    "num_layers": 10,
    "dropout": 0.5
}

# Hyperparameter update factors
hyper_param_updates = {
    "learning_rate": {
        "up": lambda lr: lr * 10,
        "down": lambda lr: lr / 10,
    },
    "batch_size": {
        "up": lambda bs: bs * 2,
        "down": lambda bs: max(bs // 2, 1),
    },
    "num_layers": {
        "up": lambda nl: nl * 2,
        "down": lambda nl: max(nl // 2, 1),
    },
    "dropout": {
        "up": lambda d: min(d + 0.1, 0.9),
        "down": lambda d: max(d - 0.1, 0.1),
    },
}

# Initial quality
best_metric_value = evaluate(hyper_params)

# Cache of hyperparameter value combinations
cache = {
    hyper_params.values(): best_metric_value
}

attempts = 1

while not done(best_metric_value, attempts):
    # Choose a hyperparameter and an update direction
    hyper_param = choice(list(hyper_params.keys()))
    update_direction = choice(["up", "down"])

    # Update the hyperparameter
    current_value = hyper_params[hyper_param]
    new_value = hyper_param_updates[hyper_param][update_direction](current_value)
    new_hyper_params = {**hyper_params, hyper_param: new_value}

    # Check if the hyperparameter value combination has been evaluated before
    if new_hyper_params.values() in cache:
        continue

    attempts += 1

    # Evaluate the new hyperparameter value combination
    metric_value = evaluate(new_hyper_params)
    cache[new_hyper_params.values()] = metric_value

    if metric_value > best_metric_value:
        best_metric_value = metric_value
        hyper_params = new_hyper_params

    print(f"Best metric value: {best_metric_value}: {hyper_params}")
```

attempts += 1

# Evaluate the new hyperparameter value combination
metric_value = evaluate(new_hyper_params)
cache[new_hyper_params.values()] = metric_value

if metric_value > best_metric_value:
    best_metric_value = metric_value
    hyper_params = new_hyper_params

print(f"Best metric value: {best_metric_value}: {hyper_params}")
Remedy: Early Stopping and Checkpointing

We can use the learned parameters from before we detected overfitting.
```python
for epoch in range(num_epochs):
    model.train()
    for X, y in train_loader:
        yhat = model(X)
        loss = criterion(y, yhat)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    model.eval()
    with torch.no_grad():
        for X, y in valid_loader:
            yhat = model(X)
            loss = criterion(y, yhat)
            metric = metrics(y, yhat, model, metric)

    if metric.is_best():
        model.save(f"model{epoch}.pkl")
```
Remedy: Regularization

We can artificially constrain the parameter magnitudes

\[ L = \frac{1}{2} (\hat{y} - y)^2 + \frac{\gamma}{2} \|\theta\|^2 \]

\[ \frac{\partial L}{\partial \theta} = X\theta + \gamma \theta \]

0 ≤ γ ≤ 1 weight decay term
\[ L = \frac{1}{2} (\hat{y} - y)^2 + \frac{\lambda}{2} ||\theta||^2 \]

\[
\frac{\partial L}{\partial \theta} = 2 \cdot \frac{1}{2} (\hat{y} - y) + \lambda \frac{\partial L}{\partial \theta} \]

Chain rule

\[
\frac{\partial y}{\partial \theta} \text{ mSE} + \lambda \theta \]

weight decay factor

\[
\hat{\theta} = \theta - \frac{\alpha}{\lambda} \left( (\frac{\partial y}{\partial \theta}) \text{ mSE} + \lambda \theta \right) \]

\[
\theta_{t+1} = \theta_t - \alpha \left( (\frac{\partial y}{\partial \theta}) \text{ mSE} + \lambda \theta \right) \]

\[
\theta = 10 - 2 = 8 \\
\hat{\theta} = 8 - (-2) = 10 \\
\Theta = -12 - (-2) = -10 \\
\theta = 10 - (-2) = 12 \\
\hat{\theta} = 12 - (-2) = 14 \\
\Theta = 14 - (-2) = 16 \\
\theta = 16 - (-2) = 18 \\
\hat{\theta} = 18 - (-2) = 20 \\
\Theta = 20 - (-2) = 22 \\
\theta = 22 - (-2) = 24 \\
\hat{\theta} = 24 - (-2) = 26 \\
\Theta = 26 - (-2) = 28 \\
\theta = 28 - (-2) = 30 \\
\hat{\theta} = 30 - (-2) = 32 \\
\Theta = 32 - (-2) = 34 \\
\theta = 34 - (-2) = 36 \\
\hat{\theta} = 36 - (-2) = 38 \\
\Theta = 38 - (-2) = 40
Remedy: Dropout

We can train the model in such a way that breaks memorization.

model.train() vs model.eval()
Remedy: Dropout

We can train the model in such a way that breaks memorization

- Randomly set neuron outputs to zero
- Choose a different set of neurons each time
- The model needs redundant representations
- This leads to more general representations
- A single pathway cannot memorize the input

```python
# In model.train() mode
for layer in model.layers():
    keep_prob = 1 - dropout_rate
    keep = torch.rand_like(layer.shape) < keep_prob
    activation *= keep.float()
    activation /= keep_prob

# In model.eval() mode
for layer in model.layers():
    activation *= 1.0
```
Remedy: Data Augmentation

for epoch in range(num_epochs):
    model.train()
    for X, y in train_loader:
        yhat = model(X)
        loss = criterion(y, yhat)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    model.eval()
    with torch.no_grad():
        for X, y in valid_loader:
            yhat = model(X)
            loss = criterion(y, yhat)
            metric = metrics(y, yhat, model)

https://albumentations.ai/
**Remedy: Domain Randomization**

- This process happens during the data synthesis/creation process.
- It often relies on simulation, and it is frequently used to cross the simulation-to-reality gap.
- This is often called Sim2Real in machine learning and robotics.

“Illustration of our approach. An object detector is trained on hundreds of thousands of low-fidelity rendered images with random camera positions, lighting conditions, object positions, and non-realistic textures. At test time, the same detector is used in the real world with no additional training.”

— Tobin et al.
Summary

• Models can accidentally memorize the input data instead of learning some useful, general property

• We can prevent overfitting/memorization with several remedies

• Most remedies try to

  • Artificially limit the magnitude of parameter values (early stopping, regularization)
  
  • Add noise and randomness to the training process (dropout, augmentation, domain randomization)

• We often use these remedies together