Regularization for Deep Learning

Deep Learning, Ian Goodfellow

Jinhwan Suk, Department of Mathematical Science, KAIST

Central problem in machine learning

Overfitting, Generalization

Regularization :

> “ *Any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.* ”

- Put extra constraints and penalties
- Ensemble method

Many regularization approaches are based on ***limiting the capacity*** *of modes by adding a parameter* ***norm penalty*** $\Omega(\theta)$ *to the objective function* J.

Parameter Norm Penalties

Regularized objective function($\tilde{J}$) :

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta),$$

where $\alpha \in [0, \infty]$.

There are many choices for the parameter norm Ω.

Ω penalizes *only the weights* of the affine transform and leaves the biases unregularized.

Parameter Norm Penalties

Why do we leave bias unregularized??

Regularization of an estimator works by trading increased bias for reduced variance.

$\Longrightarrow$ An effective regularizer is one that makes a profitable trade.

In DNN, bias do not induce too much variance.

Parameter Norm Penalties

L^2 Parameter Regularization

Weight decay(Ridge regression) : $\Omega(\theta) = \frac{1}{2}\|w\|_2^2$

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \frac{1}{2}\alpha\|w\|_2^2$$

Parameter gradient :

$$\nabla_w \tilde{J}(w; X, y) = \alpha w + \nabla_w J(w; X, y)$$

$$w \leftarrow w - \varepsilon(\alpha w + \nabla_w J(w; X, y))$$
$$= (1 - \varepsilon\alpha)w - \varepsilon \nabla_w J(w; X, y)$$

We will simplify the analysis by making a quadratic approximation to the objective function in the neighborhood of $w* = argmin_w J(w)$.

The approximation $\hat{J}(w)$ is given by

$$\hat{J}(w) = J(w*) + \nabla_w J(w*)(w - w*) + \frac{1}{2}(w - w*)^T H_{w*}(w - w*) + o(\|w - w*\|_2^2)$$
$$= J(w*) + \frac{1}{2}(w - w*)^T H_{w*}(w - w*) + o(\|w - w*\|_2^2)$$

The minimum of $\hat{J}$ occurs where its gradient

$$\nabla_w \hat{J}(w) = H_{w*}(w - w*) + o(\|w - w*\|_2)$$

is equal to ***0***.

Hence, the regularized objective function can be approximated by

$$\nabla_w \tilde{J}(w) = \alpha w + H_{w*}(w - w*) + o(\|w - w*\|_2)$$

Therefore,

$$\begin{aligned}
\nabla_w \tilde{J}(\tilde{w}) = 0 &\Leftrightarrow \alpha\tilde{w} + H_{w*}(\tilde{w} - w*) + o(\|\tilde{w} - w*\|_2) = 0 \\
&\Leftrightarrow (H_{w*} + \alpha I)\tilde{w} = H_{w*}w* + o(\|\tilde{w} - w*\|_2) = 0 \\
&\Leftrightarrow \tilde{w} = (H_{w*} + \alpha I)^{-1}H_{w*}w* + o(\|\tilde{w} - w*\|_2) \\
&\Leftrightarrow \tilde{w} = Q^T \frac{\Lambda}{\Lambda + \alpha I} Q w* + o(\|\tilde{w} - w*\|_2)
\end{aligned}$$

Remark

Given eigenvector $\overrightarrow{v}$ of Hessian H, corresponding eigenvalue λ means the curvature of the normal section along the direction $\overrightarrow{v}$.

Conclusion :

- Only directions along which the parameters contribute significantly to reducing the objective function(large eigenvalue(or, curvature)) are preserved relatively intact

- Components of the wight corresponding to unimportant directions are decayed away through the use of the regularization.

Parameter Norm Penalties

L^1 Parameter Regularization

We will discuss the effect of L^1 on the simple linear regression model.

$$J(w) = (Xw - y)^T(Xw - y);\ H = X^TX$$
$$w* = (X^TX)^{-1}Xy$$

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\|w\|_1$$
$$w* = argmin_w J(w)$$
$$\nabla_w \hat{J}(w) = H_{w*}(w - w*) + o(\|w - w*\|_2)$$

We may assume that H_{w*} is diagonal, $H_{w*} = diag([H_1, H_2, \ldots, H_n]), H_i > 0$.

(The assumption holds if features of X are uncorrelated.)

$$\hat{J}(w;X,y) = J(w^*;X,y) + \sum_i \left[\frac{1}{2}H_i(w_i - w_i^*)^2 + \alpha|w_i|\right]$$

Solution of the problem of minimizing this regularized cost function :

$$w_i = sign(w_i^*)\max\{|w_i^*| - \frac{\alpha}{H_i}, 0\}$$

Consider the situation $w_i^* > 0$.

1. $w_i^* \leq \frac{\alpha}{H_i} : w_i = 0$
2. $w_i^* > \frac{\alpha}{H_i} : w_i = w_i^* - \frac{\alpha}{H_i}$

When $w_i^* < 0$,

1. $w_i^* \geq -\frac{\alpha}{H_i} : w_i = 0$
2. $w_i^* < -\frac{\alpha}{H_i} : w_i = w_i = w_i^* + \frac{\alpha}{H_i}$

L^1 regularization results more **sparse** solution.

$\Rightarrow$ feature selection / LASSO

Norm Penalties as Constrained Optimization

Recall from section 4.4

$$\begin{aligned} \min_{\theta} \quad & J(\theta; X, y) \\ \text{s.t.} \quad & \Omega(\theta) \leq k \end{aligned}$$

Generalized Lagrange function is

$$\mathscr{L}(\theta, \alpha; X, y) = J(\theta; X, y) + \alpha(\Omega(\theta) - k)$$

and KKT conditions are

$$\begin{aligned} \Omega(\theta^*) &\leq k \\ \alpha^* &\geq 0 \\ \alpha^*(\Omega(\theta^*) - k) &= 0 \\ \nabla(\mathscr{L}(\theta^*, \alpha^*; X, y)) &= 0 \end{aligned}$$

Dataset Augmentation

Data Transformation

The best way to make a machine learning model generalize better is to train on ***more data***.

Create fake data and add it to the training set.

Data augmentation has been a particularly effective technique for a specific classification problem ; Object recognition.

- Translation
- Rotation
- Scaling

Be Careful !! : Don’t apply transformations that would change the correct class.

Data Augmentation

Injecting noise

- Neural net prove not to be very robust to noise(ICML, 2010)

- Solution :
  - Train with random noise applied to inputs(or, hidden units)
  - Injecting noise = data augmentation
  - Denoising autoencoder(2008)

Deep networks for robust visual recognition

Yichuan Tang Y3TANG@UWATERLOO.CA
Chris Eliasmith CELIASMITH@UWATERLOO.CA
Centre for Theoretical Neuroscience, University of Waterloo, Waterloo ON N2L 3G1 CANADA

Abstract

Deep Belief Networks (DBNs) are hierarchical generative models which have been used successfully to model high dimensional visual data. However, they are not robust to common variations such as occlusion and random noise. We explore two strategies for improving the robustness of DBNs. First, we show that a DBN with sparse connections in the first layer is more robust to variations that are not in the training set. Second, we develop a probabilistic denoising algorithm to determine a subset of the hidden layer nodes to unclamp. We show that this can be applied to any feedforward network classifier with localized first layer connections. Recognition results after denoising are significantly better over the standard DBN implementations for various sources of noise.

1. Introduction

Deep Belief Networks (DBNs) are hierarchical generative models with many latent variables that effectively model high dimensional visual image data (Hinton et al., 2006). A DBN is trained by a greedy layer-by-layer unsupervised learning algorithm on a series of bipartite Markov Random Field (MRF) known as a Restricted Boltzmann Machine (RBM). Fine tuning by the up-down algorithm or discriminative optimization results in a deep network capable of fast feedforward classification (Hinton & Salakhutdinov, 2006; Salakhutdinov & Hinton, 2009).

DBNs model all the pixels in the visible layer probabilistically, and as a result, are not robust to images with "noise" which are not in the training set. We include occlusions, additive noise, and "salt" noise in our definition of noise in this paper.

To improve the robustness of the DBN, we introduce a modified version of the DBN termed a sparse DBN (sDBN) where the first layer is sparsely (and locally) connected. This is in part inspired by the properties of the human visual system. It is well-established that the lower cortical levels represent the visual input in a local, sparsely connected topographical manner (Hubel & Wiesel, 1959). We show that a sDBN is more robust to noise on the MNIST (LeCun et al., 1998) dataset with noise added to the test images. We then present a denoising algorithm which combines top-down and bottom-up inputs to "fill in" the subset of hidden layer nodes which are most affected by noise. (Lee & Mumford, 2003) proposed that the human visual cortex performs hierarchical Bayesian inference where "beliefs" are propagated up and down the hierarchy. Our attention-esque top-down feedback can be thought of as a type of "belief" that helps to identify object versus non-object (noise) elements in the visible layer.

2. Related Work

Sparsely connected weights have been widely used in visual recognition algorithms (Fukushima, 1983; LeCun et al., 1998; Serre et al., 2005). Most of these algorithms contain a max-pooling stage following a convolutional stage to provide a certain amount of translational and scale invariance. Recently there has been work combining the convolutional approach with the DBN (Lee et al., 2009; Norouzi et al., 2009). These efforts enforce sparse connections similar in spirit to those enforced here. However, unlike those methods, our main motivation is not to provide translational invariance and/or to reduce the number of model parameters, but rather to diminish the effect of noise on the activations of hidden layer nodes. In addition, our algorithm does not require weight sharing (applying the same filter across an image), which would increase the total number of hidden layer nodes and increase the computational complexity of our denoising algo-

Appearing in *Proceedings of the 27th International Conference on Machine Learning*, Haifa, Israel, 2010.

Data Augmentation

A/B testing

- When comparing machine learning benchmark results, taking the effect of dataset augmentation into account is important.

- When comparing machine learning algorithm A and B, make sure that both algorithms are evaluated using the same dataset augmentation schemes.

Noise Robustness

Input noise = data augmentation

Adding noise to the weights : used in RNN

- This can be interpreted as a Bayesian inference over the weights.

Under some assumptions, noise applied to weights can be interpreted as a more traditional form of regularization.

Consider the regression setting, where we wish to train a function $\hat{y}(x)$.

$$J = \mathbb{E}_{p(x,y)}[(\hat{y}(x) - y)^2]$$

The training set consists of m labeled examples

$$\{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)})\}.$$

We now assume that with each input presentation we also include a random perturbation $\epsilon_W \sim N(\epsilon; 0, \eta I)$ of the network weights.

$$\tilde{J}_W = \mathbb{E}_{p(x,y,\epsilon_W)}[(\hat{y}_{\epsilon_W} - y)^2]$$

For small η, using Taylor's theorem,

$$\hat{y}_{\varepsilon_W}(x) = \hat{y}(x) + \nabla_W \hat{y}(x)\varepsilon + o(\|\varepsilon\|^2)$$

$$\begin{aligned}
\mathbb{E}_{p(x,y,\varepsilon_W)}[(\hat{y}_{\varepsilon_W}(x) - y)^2] &= \mathbb{E}_{p(x,y,\varepsilon_W)}[(\hat{y}_{\varepsilon_W}(x) - \hat{y}(x) + \hat{y}(x) - y)^2] \\
&= \mathbb{E}_{p(x,y,\varepsilon_W)}[(\hat{y}_{\varepsilon_W}(x) - \hat{y}(x))^2] + \mathbb{E}_{p(x,y,\varepsilon_W)}[(\hat{y}_{\varepsilon_W}(x) - \hat{y}(x))(\hat{y}(x) - y)] + \mathbb{E}_{p(x,y)}[(\hat{y}(x) - y)^2] \\
&= \mathbb{E}_{p(x,y,\varepsilon_W)}[\|\nabla_W \hat{y}(x)\|^2 \|\varepsilon\|^2] + \mathbb{E}_{p(x,y,\varepsilon_W)}[\nabla_W \hat{y}(x)\varepsilon(\hat{y}(x)) - y)] + J \\
&= \eta \mathbb{E}_{p(x,y)}[\|\nabla_W \hat{y}(x)\|^2] + J
\end{aligned}$$

Above regularization term encourages the parameters to go to regions of parameter space *where small perturbations of the wights have a relatively small influence to the output.*

= *where the model is relatively insensitive to small variations in the weights*

Injecting Noise at the Output Targets

Label Smoothing

Datasets have some number of mistakes in the y labels.

$\Rightarrow$ It can be harmful to maximize $\log p(y \mid x)$ when y is mistake.

Answer : noise on the labels

We can assume that for some small constant ϵ, the training set label y is correct with probability $1 - \epsilon$.

Original Softmax : hard targets ([0, 0, 1, 0, 0]) Never converge.

Label Smoothing : smooth targets ([$\frac{\epsilon}{k-1}, \frac{\epsilon}{k-1}, 1-\epsilon, \frac{\epsilon}{k-1}, \frac{\epsilon}{k-1}$]) prevent the pursuit of hard probability.

Semi-Supervised Learning

Labeled data + **Unlabeled data**

(inexpensive + expensive)

⇒ **considerable improvement**

Generative Models
Graph-based methods

Semi-supervised learning

From Wikipedia, the free encyclopedia

Semi-supervised learning is an approach to machine learning that combines a small amount of labeled data with a large amount of unlabeled data during training. Semi-supervised learning falls between unsupervised learning (with no labeled training data) and supervised learning (with only labeled training data).

Unlabeled data, when used in conjunction with a small amount of labeled data, can produce considerable improvement in learning accuracy. The acquisition of labeled data for a learning problem often requires a skilled human agent (e.g. to transcribe an audio segment) or a physical experiment (e.g. determining the 3D structure of a protein or determining whether there is oil at a particular location). The cost associated with the labeling process thus may render large, fully labeled training sets infeasible, whereas acquisition of unlabeled data is relatively inexpensive. In such situations, semi-supervised learning can be of great practical value. Semi-supervised learning is also of theoretical interest in machine learning and as a model for human learning.

A set of l independently identically distributed examples $x_1, \dots, x_l \in X$ with corresponding labels $y_1, \dots, y_l \in Y$ and u unlabeled examples $x_{l+1}, \dots, x_{l+u} \in X$ are processed. Semi-supervised learning combines this information to surpass the classification performance that can be obtained either by discarding the unlabeled data and doing supervised learning or by discarding the labels and doing unsupervised learning.

Semi-supervised learning may refer to either transductive learning or inductive learning.[1] The goal of transductive learning is to infer the correct labels for the given unlabeled data $x_{l+1}, \dots, x_{l+u}$ only. The goal of inductive learning is to infer the correct mapping from X to Y.

Intuitively, the learning problem can be seen as an exam and labeled data as sample problems that the teacher solves for the class as an aid in solving another set of problems. In the transductive setting, these unsolved problems act as exam questions. In the inductive setting, they become practice problems of the sort that will make up the exam.

It is unnecessary (and, according to Vapnik's principle, imprudent) to perform transductive learning by way of inferring a classification rule over the entire input space; however, in practice, algorithms formally designed for transduction or induction are often used interchangeably.

Semi-Supervised Learning

In the context of Deep Learning

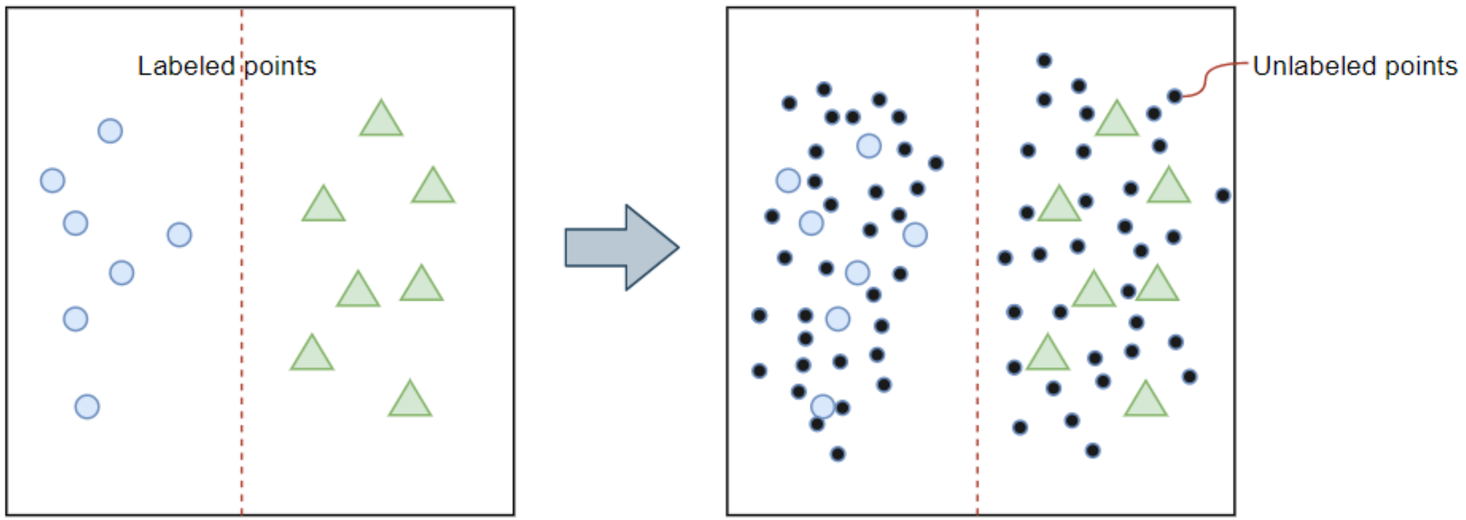
Semi-supervised learning usually refers to learning a representation

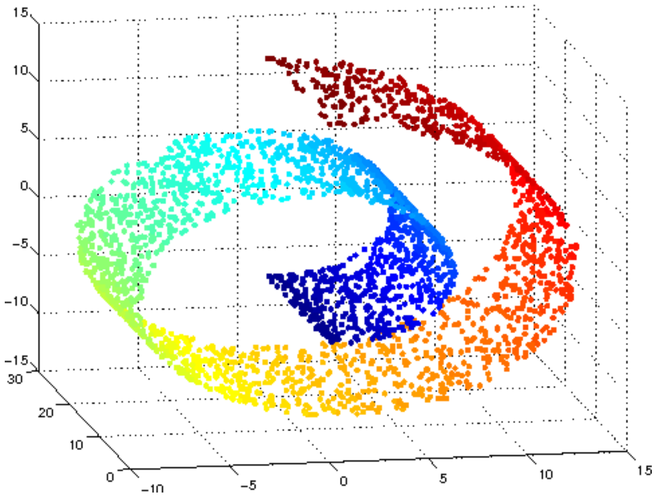
$$h = f(x)$$

*** Manifold assumption**

- The data lie approximately on a manifold of much lower dimension than input space.
- Learning f is equivalent to learning manifold.
- Same class, similar representation

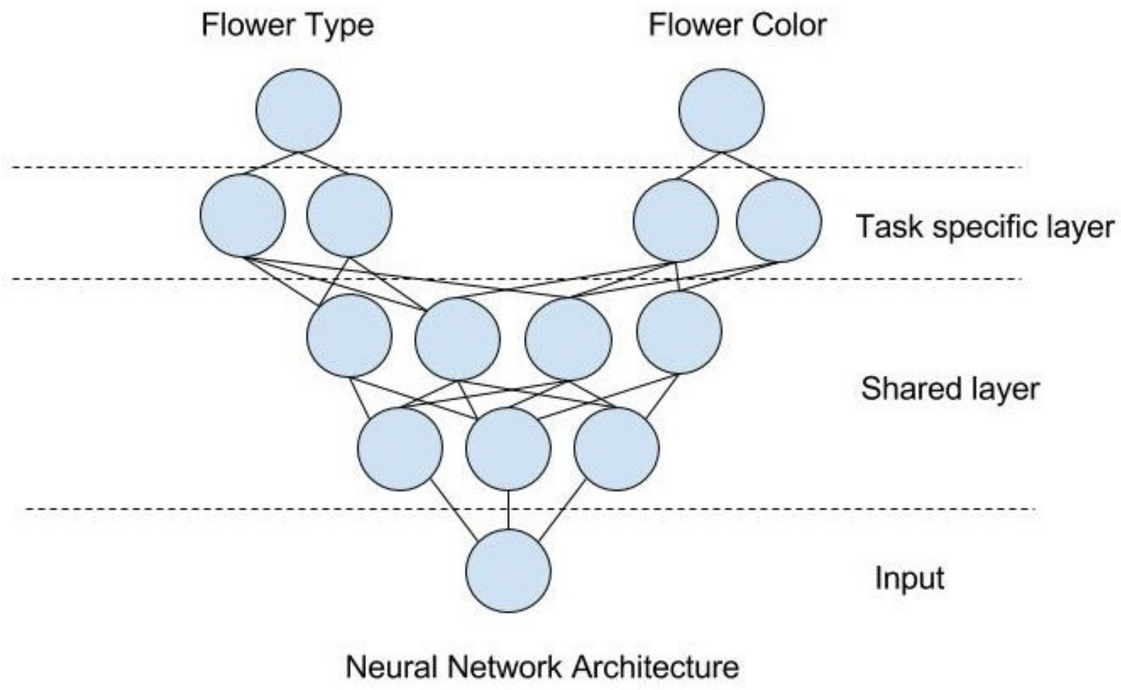
Cluster-then-label





Multitask Learning

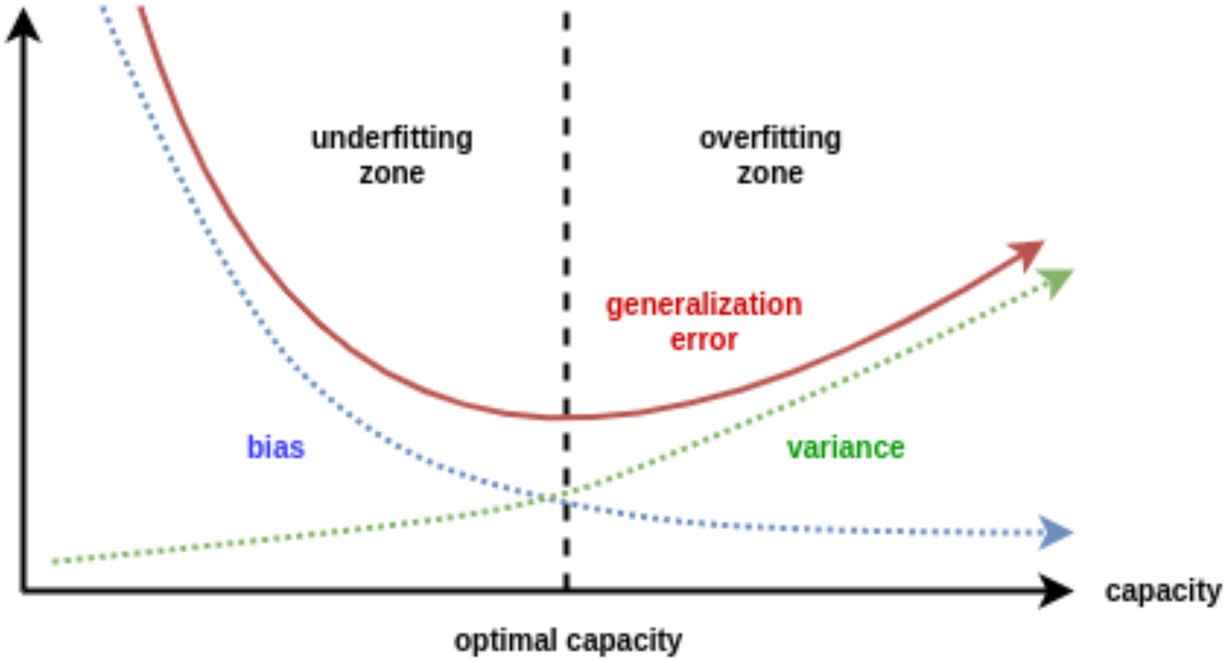
- Different supervised tasks share the same input x.
- Improved generalization(Boxter, 1995)
- *There is something shared across tasks*

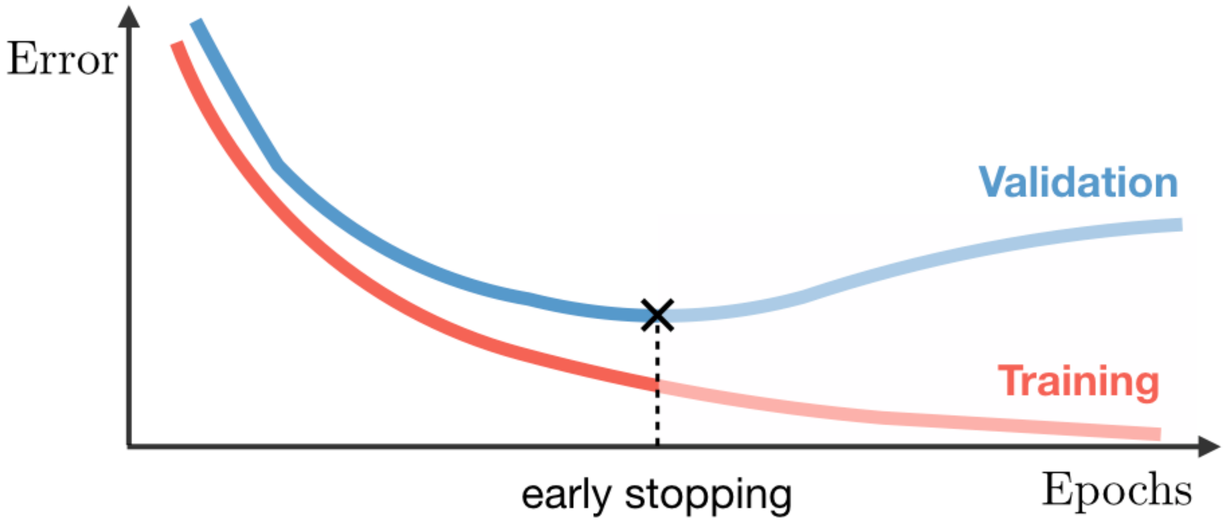


Neural Network Architecture

Early Stopping

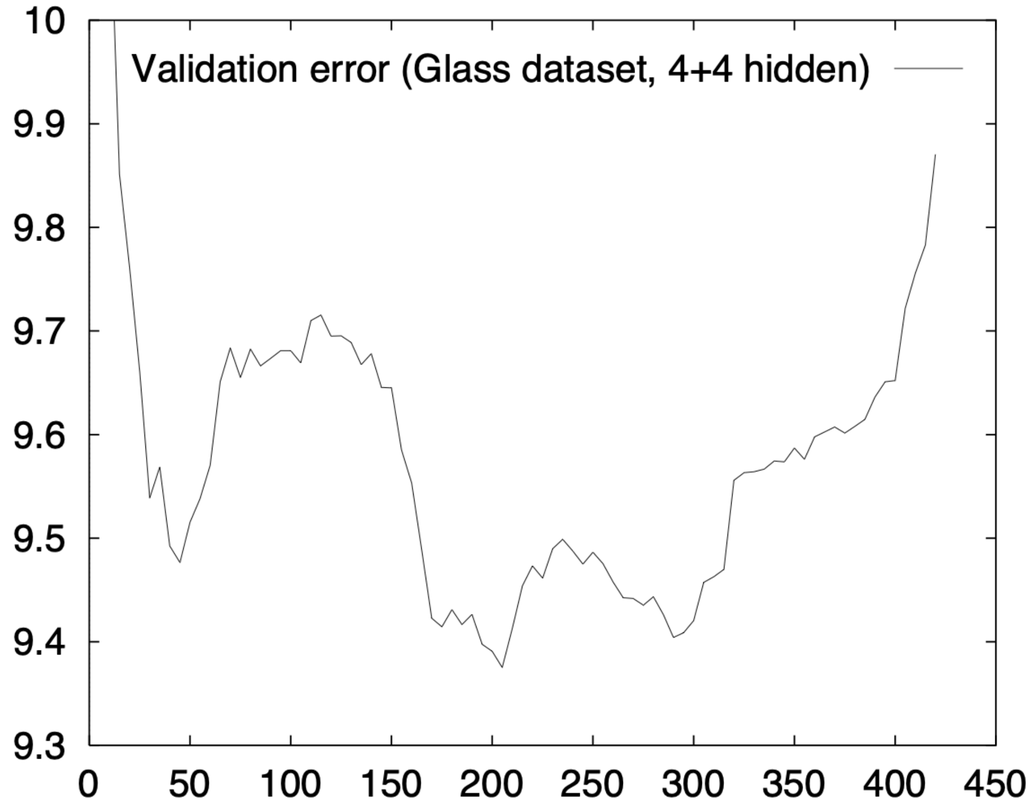
- When do we stop training??
  - Run the validation set evaluation periodically during training
  - Computational cost
- Early stopping requires a validation set





Early Stopping

But When??



Early Stopping — but when?

Lutz Prechelt (prechelt@ira.uka.de)

Fakultät für Informatik; Universität Karlsruhe
D-76128 Karlsruhe; Germany

Abstract. Validation can be used to detect when overfitting starts during supervised training of a neural network; training is then stopped before convergence to avoid the overfitting ("early stopping"). The exact criterion used for validation-based early stopping, however, is usually chosen in an ad-hoc fashion or training is stopped interactively. This trick describes how to select a stopping criterion in a systematic fashion; it is a trick for either speeding learning procedures or improving generalization, whichever is more important in the particular situation. An empirical investigation on multi-layer perceptrons shows that there exists a tradeoff between training time and generalization: From the given mix of 1296 training runs using different 12 problems and 24 different network architectures I conclude slower stopping criteria allow for small improvements in generalization (here: about 4% on average), but cost *much* more training time (here: about factor 4 longer on average).

1 Early stopping is not quite as simple

1.1 Why early stopping?

When training a neural network, one is usually interested in obtaining a network with optimal generalization performance. However, all standard neural network architectures such as the fully connected multi-layer perceptron are prone to overfitting [10]: While the network *seems* to get better and better, i.e., the error on the training set decreases, at some point during training it actually begins to get worse again, i.e., the error on unseen examples increases. The idealized expectation is that during training the generalization error of the network evolves as shown in Figure 1. Typically the generalization error is estimated by a validation error, i.e., the average error on a *validation set*, a fixed set of examples not from the training set.

There are basically two ways to fight overfitting: reducing the number of dimensions of the parameter space or reducing the effective size of each dimension. Techniques for reducing the number of parameters are greedy constructive learning [7], pruning [5, 12, 14], or weight sharing [18]. Techniques for reducing the size of each parameter dimension are regularization, such as weight decay [13] and others [25], or early stopping [17]. See also [8, 20] for an overview and [9] for an experimental comparison.

Early stopping is widely used because it is simple to understand and implement and has been reported to be superior to regularization methods in many cases, e.g. in [9].

Early stopping meta algorithm

Let n be the number of steps between evaluations.
Let p be the "patience," the number of times to observe worsening validation set error before giving up.
Let $\boldsymbol{\theta}_o$ be the initial parameters.
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_o$
$i \leftarrow 0$
$j \leftarrow 0$
$v \leftarrow \infty$
$\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
$i^* \leftarrow i$
while $j < p$ **do**
  Update $\boldsymbol{\theta}$ by running the training algorithm for n steps.
  $i \leftarrow i + n$
  $v' \leftarrow \text{ValidationSetError}(\boldsymbol{\theta})$
  if $v' < v$ **then**
    $j \leftarrow 0$
    $\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
    $i^* \leftarrow i$
    $v \leftarrow v'$
  else
    $j \leftarrow j + 1$
  end if
end while
Best parameters are $\boldsymbol{\theta}^*$, best number of training steps is i^*

Algorithm determines the best amount of time to train. The meta algorithm is a general strategy that works well with a variety of training algorithms and ways of quantifying error on the validation set.

Early Stopping

How early stopping acts as a regularizer :

Early stopping is equivalent to L^2 regularization.

$$\theta = w; \;\; w^* = argmin\, J(w)$$

$$\hat{J}(\theta) = J(w^*) + \frac{1}{2}(w - w^*)^T H_{w^*}(w - w^*) + o(\|w - w^*\|^2)$$

$$\nabla_w \hat{J}(w) = H_{w^*}(w - w^*)$$

Set $w^{(0)} = 0$; $\tau =$ optimization step ; $\epsilon =$ learning rate.

$$\begin{aligned} w^{(\tau)} &= w^{(\tau-1)} - \epsilon \nabla_w \hat{J}(w^{(\tau-1)}) \\ &= w^{(\tau-1)} - \epsilon H_{w^*}(w^{(\tau-1)} - w^*) \\ w^{(\tau)} - w^* &= (I - \epsilon H)(w^{(\tau-1)} - w^*) \end{aligned}$$

$$w^{(\tau)} - w^* = (I - \epsilon Q \Lambda Q^T)(w^{(\tau-1)} - w^*)$$

$$Q^T(w^{(\tau)} - w^*) = (I - \epsilon \Lambda) Q^T (w^{(\tau-1)} - w^*)$$

Assume that ϵ is chosen to be small enough to guarantee $|1 - \epsilon\lambda_i| < 1$.

Then,

$$Q^T w^{(\tau)} = [I - (I - \epsilon\Lambda)^\tau] Q^T w^*.$$

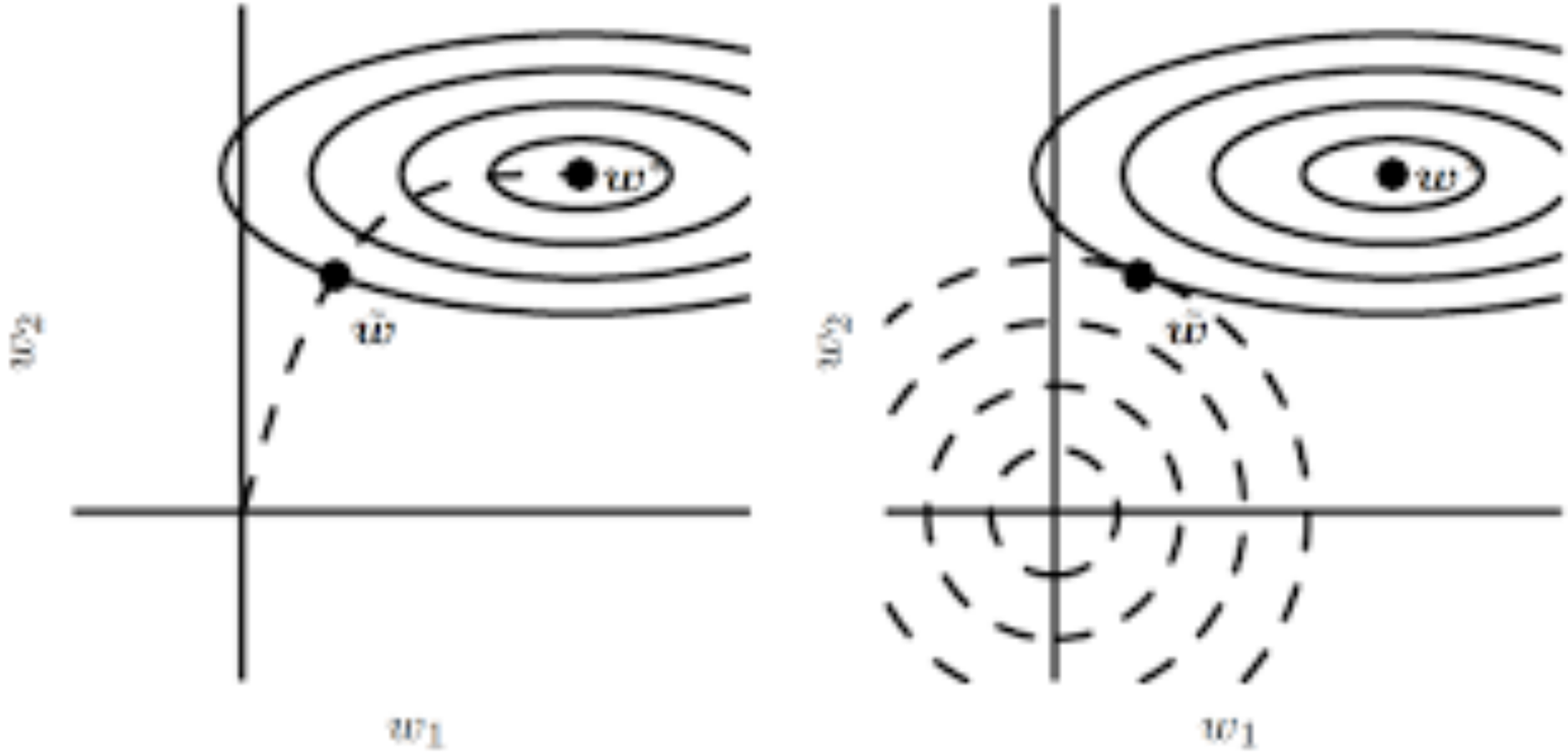
Recall the expression for $\tilde{w}$ for L^2 regularization.

$$\begin{aligned} Q^T \tilde{w} &= (\Lambda + \alpha I)^{-1} Q^T w^* \\ &= [I - (\Lambda + \alpha I)^{-1}\alpha] Q^T w^* \end{aligned}$$

If the hyper parameters ϵ, α, and τ are chosen such that

$$(I - \epsilon\Lambda)^\tau = (\Lambda + \alpha I)^{-1}\alpha,$$

then L^2 regularization and early stopping can be seen as equivalent.


w
$\tilde{w}$
w_2
w_1
w
$\tilde{w}$
w_2
w_1

Define the Network

Defining a simple MLP model.

In [3]:
```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, 10)
        self.dropout = nn.Dropout(0.5)
    def forward(self, x):
        # flatten image input
        x = x.view(-1, 28 * 28)
        # add hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        # add hidden layer, with relu activation function
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        # add output layer
        x = self.fc3(x)
        return x

# initialize the NN
model = Net()
print(model)
```

```
Net(
  (fc1): Linear(in_features=784, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=10, bias=True)
  (dropout): Dropout(p=0.5)
)
```

Specify Loss Function and Optimizer

In [4]:
```
# specify loss function
criterion = nn.CrossEntropyLoss()

# specify optimizer
optimizer = torch.optim.Adam(model.parameters())
```

Import the Early Stopping Class

In [5]:
```
# import EarlyStopping
from pytorchtools import EarlyStopping
```

Train the Model using Early Stopping

In [6]:
```
def train_model(model, batch_size, patience, n_epochs):

    # to track the training loss as the model trains
    train_losses = []
    # to track the validation loss as the model trains
    valid_losses = []
    # to track the average training loss per epoch as the model trains
    avg_train_losses = []
    # to track the average validation loss per epoch as the model trains
    avg_valid_losses = []

    # initialize the early_stopping object
    early_stopping = EarlyStopping(patience=patience, verbose=True)

    for epoch in range(1, n_epochs + 1):

        ###################
        # train the model #
        ###################
        model.train() # prep model for training
        for batch, (data, target) in enumerate(train_loader, 1):
            # clear the gradients of all optimized variables
            optimizer.zero_grad()
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the loss
            loss = criterion(output, target)
            # backward pass: compute gradient of the loss with respect to model parameters
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()
            # record training loss
            train_losses.append(loss.item())
```

```
        ######################
        # validate the model #
        ######################
        model.eval() # prep model for evaluation
        for data, target in valid_loader:
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the loss
            loss = criterion(output, target)
            # record validation loss
            valid_losses.append(loss.item())

        # print training/validation statistics
        # calculate average loss over an epoch
        train_loss = np.average(train_losses)
        valid_loss = np.average(valid_losses)
        avg_train_losses.append(train_loss)
        avg_valid_losses.append(valid_loss)

        epoch_len = len(str(n_epochs))

        print_msg = (f'[{epoch:>{epoch_len}}/{n_epochs:>{epoch_len}}] ' +
                     f'train_loss: {train_loss:.5f} ' +
                     f'valid_loss: {valid_loss:.5f}')

        print(print_msg)

        # clear lists to track next epoch
        train_losses = []
        valid_losses = []

        # early_stopping needs the validation loss to check if it has decresed,
        # and if it has, it will make a checkpoint of the current model
        early_stopping(valid_loss, model)

        if early_stopping.early_stop:
            print("Early stopping")
            break

    # load the last checkpoint with the best model
    model.load_state_dict(torch.load('checkpoint.pt'))

    return  model, avg_train_losses, avg_valid_losses
```

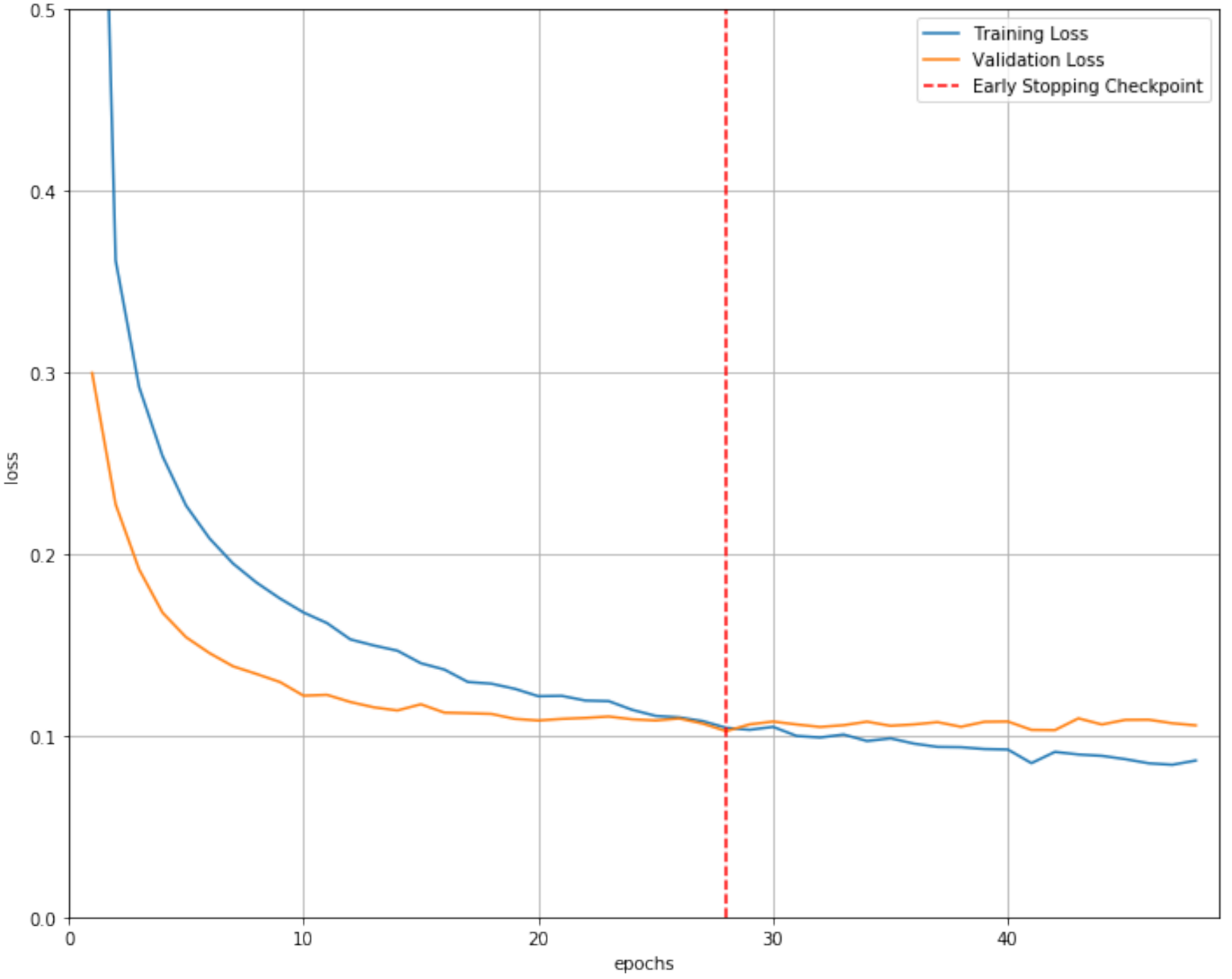
In [7]:
```
batch_size = 256
n_epochs = 100

train_loader, test_loader, valid_loader = create_datasets(batch_size)

# early stopping patience; how long to wait after last time validation loss improved.
patience = 20

model, train_loss, valid_loss = train_model(model, batch_size, patience, n_epochs)
```

```
[  1/100] train_loss: 0.84499 valid_loss: 0.29977
[  2/100] train_loss: 0.36182 valid_loss: 0.22742
Validation loss decreased (inf --> 0.227419).  Saving model ...
[  3/100] train_loss: 0.29205 valid_loss: 0.19163
Validation loss decreased (0.227419 --> 0.191628).  Saving model ...
[  4/100] train_loss: 0.25390 valid_loss: 0.16771
Validation loss decreased (0.191628 --> 0.167714).  Saving model ...
[  5/100] train_loss: 0.22671 valid_loss: 0.15422
Validation loss decreased (0.167714 --> 0.154222).  Saving model ...
[  6/100] train_loss: 0.20862 valid_loss: 0.14546
Validation loss decreased (0.154222 --> 0.145459).  Saving model ...
[  7/100] train_loss: 0.19482 valid_loss: 0.13821
Validation loss decreased (0.145459 --> 0.138206).  Saving model ...
[  8/100] train_loss: 0.18431 valid_loss: 0.13398
Validation loss decreased (0.138206 --> 0.133979).  Saving model ...
[  9/100] train_loss: 0.17554 valid_loss: 0.12953
Validation loss decreased (0.133979 --> 0.129535).  Saving model ...
[ 10/100] train_loss: 0.16785 valid_loss: 0.12202
Validation loss decreased (0.129535 --> 0.122023).  Saving model ...
[ 11/100] train_loss: 0.16202 valid_loss: 0.12249
EarlyStopping counter: 1 out of 20
[ 12/100] train_loss: 0.15300 valid_loss: 0.11852
Validation loss decreased (0.122023 --> 0.118516).  Saving model ...
[ 13/100] train_loss: 0.14965 valid_loss: 0.11560
Validation loss decreased (0.118516 --> 0.115598).  Saving model ...
[ 14/100] train_loss: 0.14680 valid_loss: 0.11387
Validation loss decreased (0.115598 --> 0.113867).  Saving model ...
[ 15/100] train_loss: 0.13988 valid_loss: 0.11728
EarlyStopping counter: 1 out of 20
[ 16/100] train_loss: 0.13641 valid_loss: 0.11269
Validation loss decreased (0.113867 --> 0.112686).  Saving model ...
[ 17/100] train_loss: 0.12957 valid_loss: 0.11237
Validation loss decreased (0.112686 --> 0.112374).  Saving model ...
EarlyStopping counter: 3 out of 20
[ 34/100] train_loss: 0.09704 valid_loss: 0.10770
EarlyStopping counter: 6 out of 20
[ 35/100] train_loss: 0.09850 valid_loss: 0.10542
EarlyStopping counter: 7 out of 20
[ 36/100] train_loss: 0.09561 valid_loss: 0.10619
EarlyStopping counter: 8 out of 20
[ 37/100] train_loss: 0.09381 valid_loss: 0.10745
EarlyStopping counter: 9 out of 20
[ 38/100] train_loss: 0.09363 valid_loss: 0.10487
EarlyStopping counter: 10 out of 20
[ 39/100] train_loss: 0.09263 valid_loss: 0.10763
EarlyStopping counter: 11 out of 20
[ 40/100] train_loss: 0.09234 valid_loss: 0.10778
EarlyStopping counter: 12 out of 20
[ 41/100] train_loss: 0.08485 valid_loss: 0.10319
EarlyStopping counter: 13 out of 20
[ 42/100] train_loss: 0.09105 valid_loss: 0.10305
EarlyStopping counter: 14 out of 20
[ 43/100] train_loss: 0.08963 valid_loss: 0.10952
EarlyStopping counter: 15 out of 20
[ 44/100] train_loss: 0.08887 valid_loss: 0.10615
EarlyStopping counter: 16 out of 20
[ 45/100] train_loss: 0.08704 valid_loss: 0.10870
EarlyStopping counter: 17 out of 20
[ 46/100] train_loss: 0.08477 valid_loss: 0.10877
EarlyStopping counter: 18 out of 20
[ 47/100] train_loss: 0.08397 valid_loss: 0.10682
EarlyStopping counter: 19 out of 20
[ 48/100] train_loss: 0.08630 valid_loss: 0.10565
EarlyStopping counter: 20 out of 20
Early stopping
```


Training Loss
Validation Loss
Early Stopping Checkpoint
loss
epochs
0.0
0.1
0.2
0.3
0.4
0.5
0
10
20
30
40

Thank you.