Deep Learning - Foundations and Concepts

Chapter 9. Regularization

nonlineark@github

March 5, 2025

Outline

Inverse problems

- Most machine learning tasks are examples of inverse problems:
  - Forward problem: Given a conditional distribution $p(t|x)$ along with a finite set of input points $\{x_1, \ldots, x_N\}$, sample corresponding values $\{t_1, \ldots, t_N\}$ from that distribution.
  - Inverse problem: Infer a distribution given only a finite number of samples.
- We need a way to choose a specific distribution from amongst the infinitely many possibilities. The preference for one choice over others is called inductive bias or prior knowledge:
  - Small changes in the input values should lead to small changes in the output values: The sum-of-squares regularizer.
  - When detecting objects in images, there should be translation invariance: Convolutional neural network.
  - Additional data from a different, but related, task can be used to help determine the learnable parameters in a neural network: Transfer learning and multi-task learning.

No free lunch theorem

The no free lunch theorem: Every learning algorithm is as good as any other when averaged over all possible problems:

- Even very flexible neural networks contain important inductive biases, it is not possible to learn purely from data in the absence of any bias.
- In trying to find general-purpose learning algorithms, we are really seeking inductive biases that are appropriate to the broad classes of applications that will be encountered in practice.
- Inductive biases can be incorporated through:
  - The form of distribution.
  - The addition of a regularization term to the error function used during training.
  - The training process.

Symmetry and invariance

- In many applications of machine learning, the predictions should be unchanged under one or more transformations of the input variables:
  - Translation invariance.
  - Scale invariance.
- Transformations that leave particular properties unchanged represent symmetries. The set of all transformations corresponding to a particular symmetry form a group.

Symmetry and invariance

Efficient approaches for encouraging an adaptive model to exhibit the required invarances:

- Pre-processing: Invariances are built into a pre-processing stage by computing features of the data that are invariant under the required transformations.
- Regularized error function: A regularization term is added to the error function to penalize changes in the model output when the input is subject to one of the invariant transformations.
- Data augmentation: The training set is expanded using replicas of the training data points, transformed according to the desired invariances and carrying the same output target values as the untransformed examples.
- Network architecture: The invariance properties are built into the structure of a neural network through an appropriate choice of network architecture.

Equivariance

A generalization of invariance is called equivariance in which the output of the network, instead of remaining constant when the input is transformed, is itself transformed in a specific way:

$$\mathcal{S}(\mathcal{T}(I)) = \tilde{\mathcal{T}}(\mathcal{S}(I))$$

For example:

- $\mathcal{S}$: Measures the orientation of an object within an image.
- $\mathcal{T}$: Rotation.
- $\tilde{\mathcal{T}}$: Increase or decrease the scalar orientation.

Weight decay

- The simplest regularizer comprises the sum of the squares of the model parameters: $\tilde{E}(w) = E(w) + \frac{\lambda}{2} w^T w$.
- The sum-of-squares regularizer is known as weight decay because it encourages weight values to decay towards zero.
- The sum-of-squares regularizer can be interpreted as the negative logarithm of a zero-mean Gaussian prior distribution over the weight vector w, which is a way to include prior knowledge into the model training process.
- It is trivial to evaluate its derivatives for use in gradient descent training: $\nabla \tilde{E}(w) = \nabla E(w) + \lambda w$.

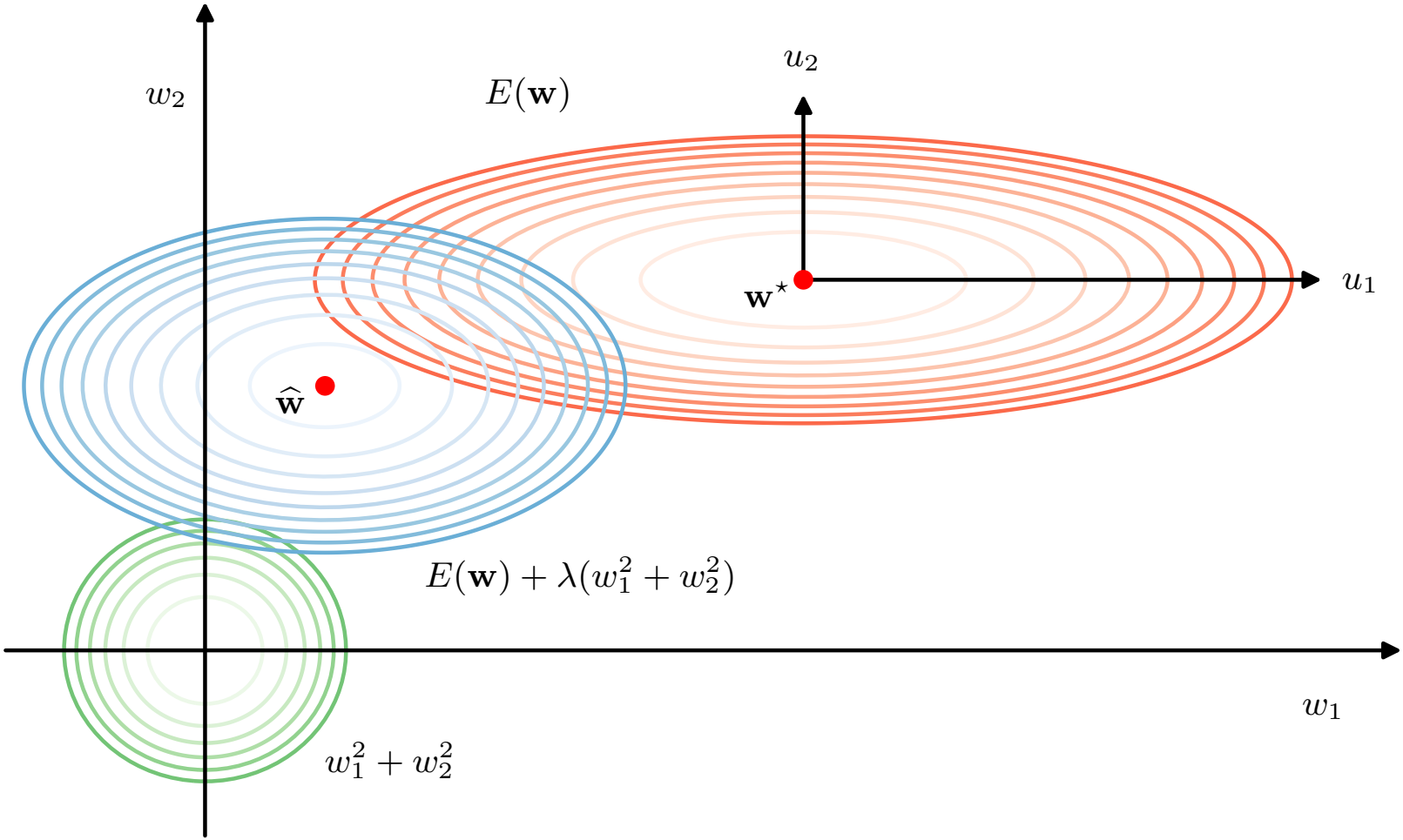
Weight decay

Consider a two-dimensional parameter space along with a sum-of-squares error function $E(w)$:

- The effect of the regularization term is to shrink the magnitudes of the weight parameters.
- The effect is much larger for the parameter corresponds to the smaller eigenvalue (here w_1) compared to that of the parameter corresponds to the larger eigenvalue (here w_2).
- The effective number of parameters is the number that remain active after regularization.
- Controlling model complexity by regularization has similarities to controlling model complexity by limiting the number of parameters:
  - $\lambda \to \infty$: The effective number of parameters is zero.
  - $\lambda = 0$: The effective number of parameters equals the total number of parameters in the model.

Weight decay

Figure: Effect of a quadratic regularizer



Consistent regularizers

Consider a two-layer network of the form:

$$z_j = h(\sum_i w_{ji} x_i + w_{j0}) \qquad y_k = \sum_j w_{kj} z_j + w_{k0}$$

Observe that linear transformations of the input data or output variables can be balanced by making a corresponding linear transformation of the weights and biases so that the mapping performed by the network is unchanged:

$$x_i \to \tilde{x}_i = a x_i + b \qquad y_k \to \tilde{y}_k = c y_k + d$$

$$w_{ji} \to \tilde{w}_{ji} = \frac{1}{a} w_{ji} \qquad w_{kj} \to \tilde{w}_{kj} = c w_{kj}$$

$$w_{j0} \to \tilde{w}_{j0} = w_{j0} - \frac{b}{a} \sum_i w_{ji} \qquad w_{k0} \to \tilde{w}_{k0} = c w_{k0} + d$$

Consistent regularizers

Clearly, the simple sum-of-squares weight decay does not have this consistency. We look for a regularizer that is invariant to re-scaling of the weights and to shifts of the biases:

$$\frac{\lambda_1}{2}\sum_{w\in\mathcal{W}_1} w^2 + \frac{\lambda_2}{2}\sum_{w\in\mathcal{W}_2} w^2$$

where $\mathcal{W}_1$ denotes the set of weights in the first layer, $\mathcal{W}_2$ denotes the set of weights in the second layer. Because biases are excluded from the summations, the prior distribution over the parameters is improper:

$$p(w;\lambda_1,\lambda_2) \propto \exp(-\frac{\lambda_1}{2}\sum_{w\in\mathcal{W}_1} w^2 - \frac{\lambda_2}{2}\sum_{w\in\mathcal{W}_2} w^2)$$

It is therefore common to include separate priors for the biases (which then break the shift invariance) that have their own hyperparameters.

Consistent regularizers

More generally, we can consider regularizers in which the weights are divided into any number of groups $\mathcal{W}_k$ so that:

$$\Omega(w) = \frac{1}{2}\sum_k \lambda_k ||w||_k^2 = \frac{1}{2}\sum_k \lambda_k \sum_{w \in \mathcal{W}_k} w^2$$

For example, we could use a different regularizer for each layer in a multilayer network.

Generalized weight decay

Generalization of the simple quadratic regularizer:

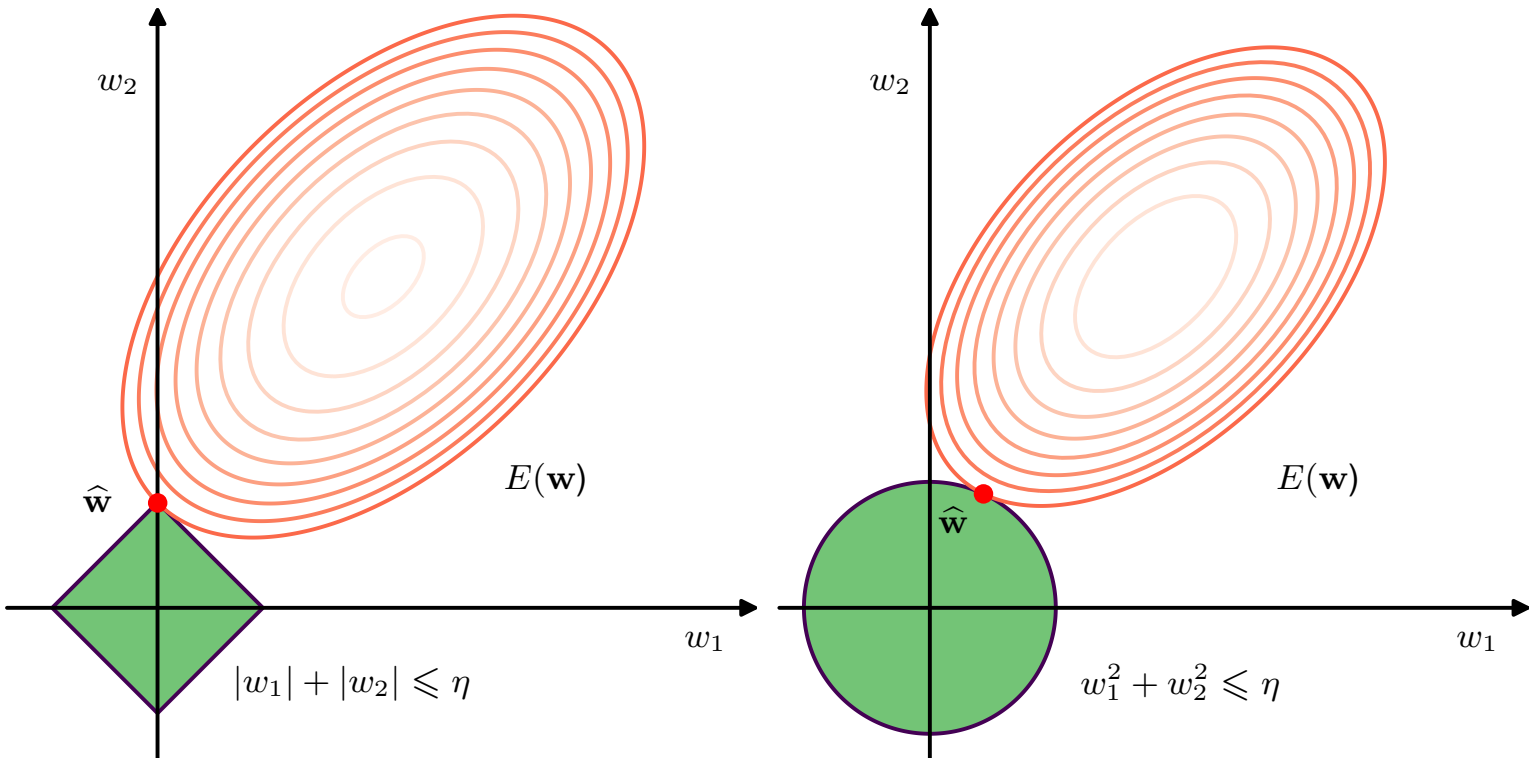
$$\Omega(w) = \frac{\lambda}{2} \sum_{m=1}^{M} |w_m|^q$$

where:

- $q = 2$ corresponds to the quadratic regularizer.
- $q = 1$ is known as a lasso. For quadratic error functions, it has the property that if λ is sufficiently large, some of the coefficients w_j are driven to zero, leading to a sparse model.

Generalized weight decay

Figure: Lasso vs. quadratic regularizer



Learning curves

During optimization of the error function through gradient descent:

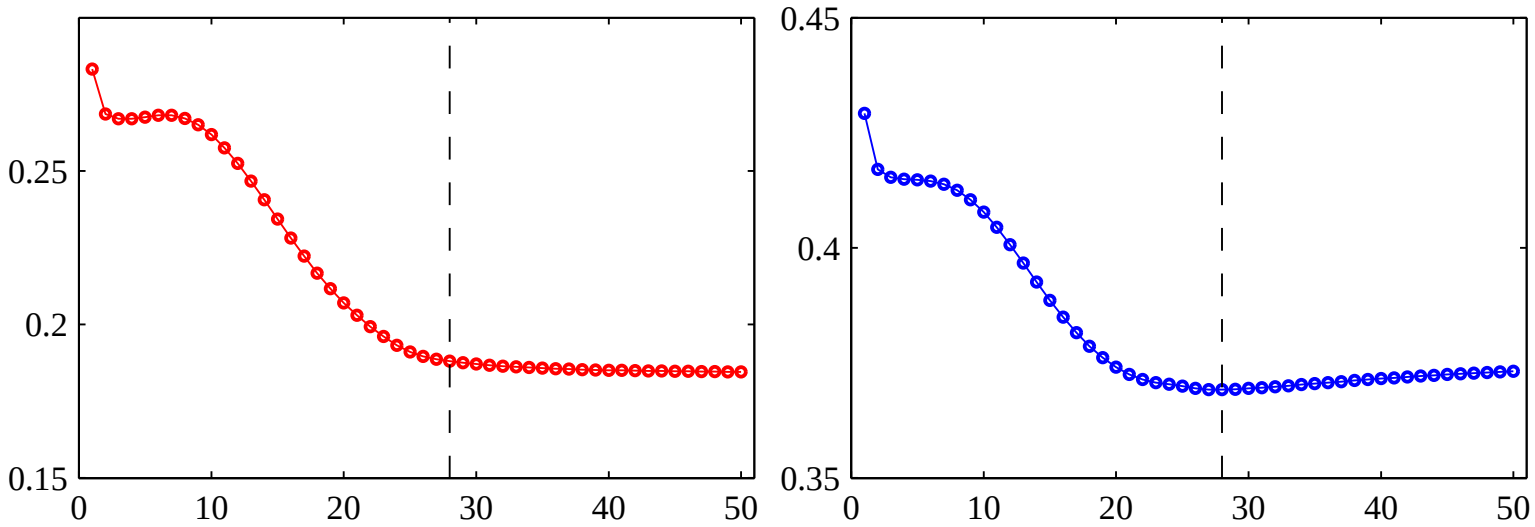
- The training error typically decreases as the model parameters are updated.
- The error for validation data may be non-monotonic.

This behavior can be visualized using learning curves, which provide insight into the progress of training and also offer a practical methodology for controlling the effective model complexity.

Early stopping

To obtain a network with good generalization performance, training should be stopped at the point of smallest error with respect to the validation data set.

Figure: An illustration of the behavior of training set error and validation set error during a typical training session

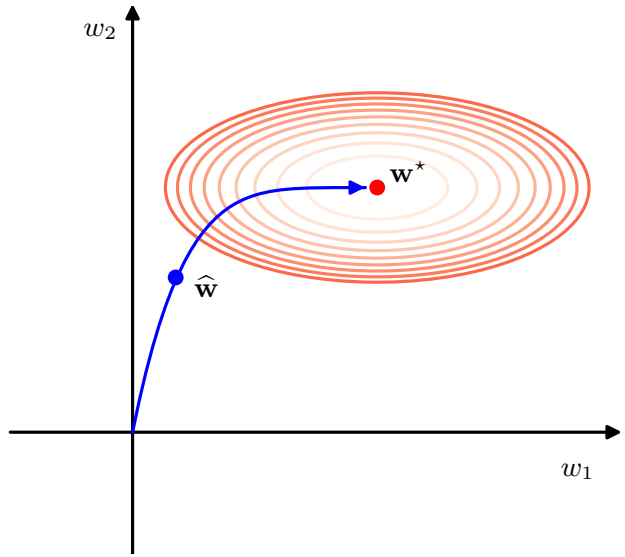


Early stopping

This behavior of the learning curves is sometimes explained qualitatively in terms of the effective number of parameters in the network:

- This number starts out small and grows during training.
- Early stopping is a way to limit the effective network complexity.

Figure: Early stopping can give similar results to weight decay for a quadratic error function



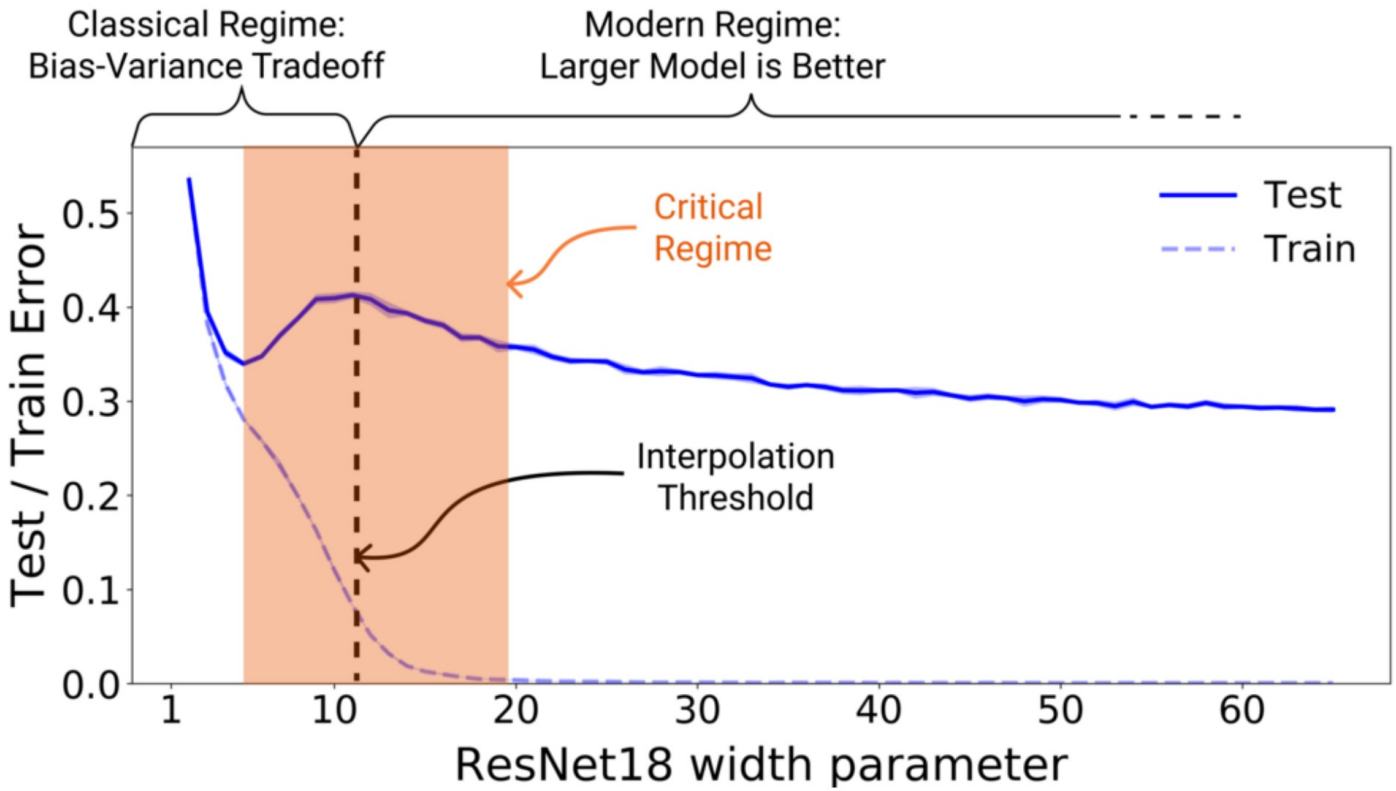
Double descent

Modern deep neural networks defeat the conventional belief:

- Conventional belief: The number of parameters in the model needs to be limited according to the size of the data set and that for a given training data set, very large models are expected to have poor performance.
- The general wisdom in the deep learning community is that bigger models are better.

Double descent

Figure: Plot of training set and test set errors for a large neural network model versus the complexity of the model



Double descent

The surprising behavior exhibits two different regimes of model fitting:

- The classical bias-variance trade-off for small to medium complexity.
- Followed by a further reduction in test set error as we enter a regime of very large models.
- The transition between the two regimes occurs roughly when the model is complex enough that the model is able to fit the training data exactly:
  - Effective model complexity: The maximum size of training data set on which a model can achieve zero training error.
  - Double descent arises when the effective model complexity exceeds the number of data points in the training set.
  - It is possible to operate in a regime where increasing the size of the training data set could actually reduce performance.

Parameter sharing

Weight sharing (parameter sharing or parameter tying):

- Impose hard constraints on the weights by forming them into groups and requiring that all weights within each group share the same value.
- Because the number of degrees of freedom is smaller than the number of connections in the network, weight sharing reduces network complexity.
- Parameter sharing is extensively used in convolutional neural networks.

Soft weight sharing

We can encourage the weight values to form several groups by considering a probability distribution that is a mixture of Gaussians:

$$p(w) = \prod_i \sum_{k=1}^{K} \pi_k \mathcal{N}(w_i; \mu_k, \sigma_k^2)$$

where K is the number of components in the mixture. Taking the negative logarithm then leads to a regularization function of the form:

$$\Omega(w) = -\sum_i \log \sum_{k=1}^{K} \pi_k \mathcal{N}(w_i; \mu_k, \sigma_k^2)$$

The total error function is then given by:

$$\tilde{E}(w) = E(w) + \lambda \Omega(w)$$

where λ is the regularization coefficient.

Soft weight sharing

The error $\tilde{E}(w)$ is minimized jointly with respect to the weights $\{w_i\}$ and with respect to the parameters $\{\pi_k, \mu_k, \sigma_k\}$ of the mixture model, which requires that we evaluate the derivatives of $\Omega(w)$ with respect to all the learnable parameters.

First, let's introduce:

$$\gamma_k(w_i) = \frac{\pi_k \mathcal{N}(w_i; \mu_k, \sigma_k^2)}{\sum_j \pi_j \mathcal{N}(w_i; \mu_j, \sigma_j^2)}$$

We could understand $\gamma_k(w_i)$ this way:

- π_k is the prior for the kth component $\mathcal{C}_k$: $\pi_k = p(\mathcal{C}_k)$.
- $\mathcal{N}(w_i; \mu_k, \sigma_k^2) = p(w_i|\mathcal{C}_k)$.
- Then $\gamma_k(w_i)$ is the posterior for $\mathcal{C}_k$: $\gamma_k(w_i) = p(\mathcal{C}_k|w_i)$.

Soft weight sharing

Now let's calculate the derivatives:

$$\frac{\partial \tilde{E}}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda \sum_k \gamma_k(w_i) \frac{w_i - \mu_k}{\sigma_k^2}$$

$$\pi_k = \frac{\exp(\eta_k)}{\sum_j \exp(\eta_j)} \qquad \frac{\partial \tilde{E}}{\partial \eta_k} = \lambda \sum_i (\pi_k - \gamma_k(w_i))$$

$$\frac{\partial \tilde{E}}{\partial \mu_k} = \lambda \sum_i \gamma_k(w_i) \frac{\mu_k - w_i}{\sigma_k^2}$$

$$\sigma_k^2 = \exp(\xi_k) \qquad \frac{\partial \tilde{E}}{\partial \xi_k} = \frac{\lambda}{2} \sum_i \gamma_k(w_i)(1 - \frac{(w_i - \mu_k)^2}{\sigma_k^2})$$

Soft weight sharing

When minimizing $\tilde{E}(w)$, we are driving all the derivatives towards zero. We see that each w_i is pulled towards the center of the kth Gaussian, with a force proportional to the posterior probability of that Gaussian for w_i. And for $\{\pi_k, \mu_k, \sigma_k\}$:

$$\pi_k \rightarrow \frac{1}{W} \sum_i \gamma_k(w_i)$$

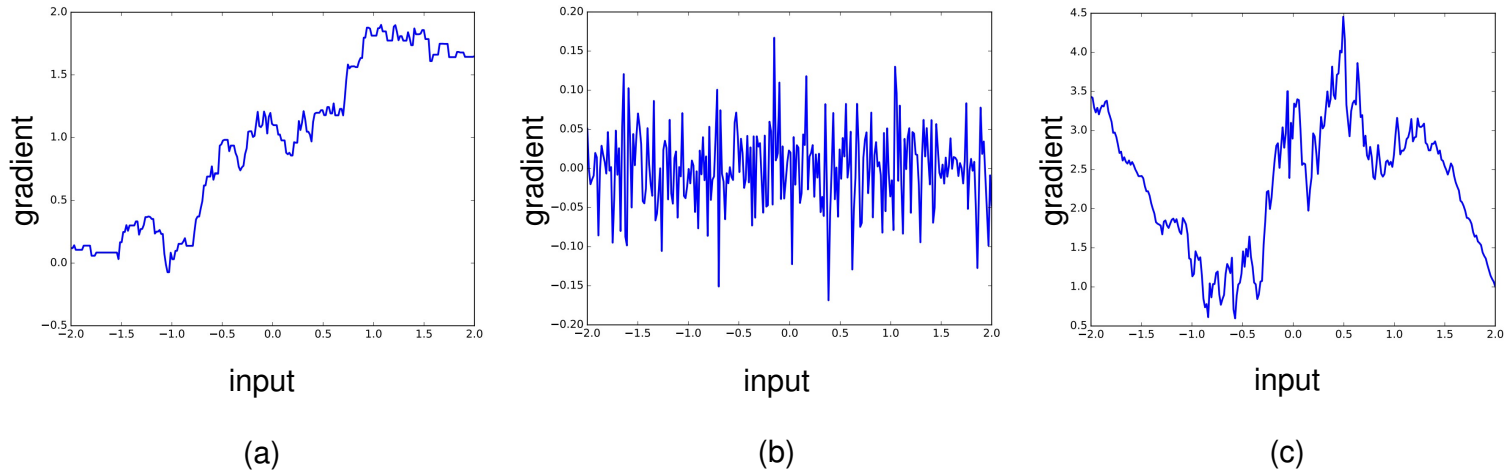
$$\mu_k \rightarrow \frac{\sum_i \gamma_k(w_i) w_i}{\sum_i \gamma_k(w_i)}$$

$$\sigma_k^2 \rightarrow \frac{\sum_i \gamma_k(w_i)(w_i - \mu_k)^2}{\sum_i \gamma_k(w_i)}$$

Shattered gradients

- The representational capabilities of neural networks increase exponentially with depth. With ReLU activation functions, there is an exponential increase in the number of linear regions that the network can represent.
- However, a consequence of this is a proliferation of discontinuities in the gradient of the error function.

Figure: Jacobian for networks with a single input and a single output



Residual connections

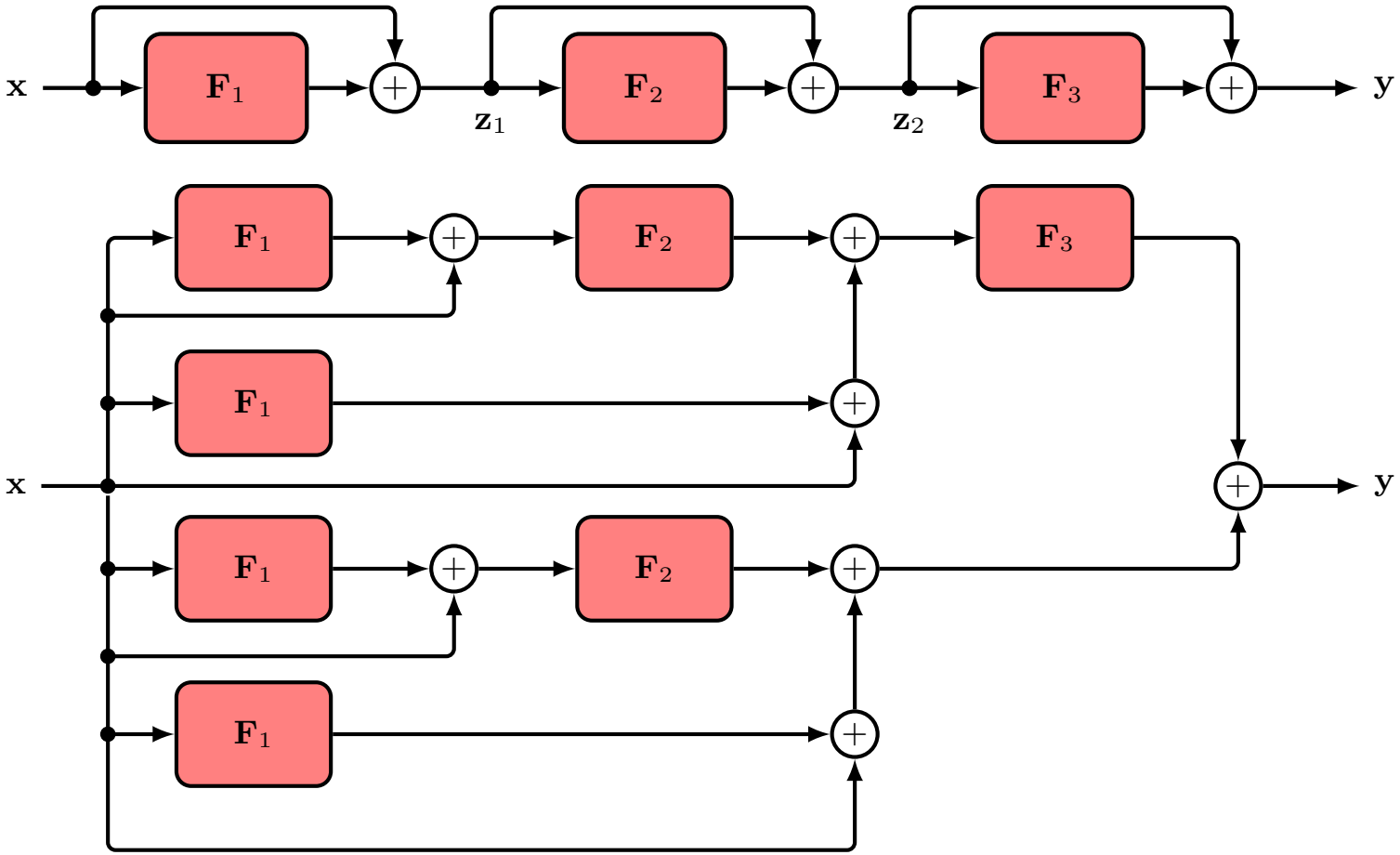
An important modification to the architecture of neural networks that greatly assists in training very deep networks is that of residual connections, which consist simply of adding the input to each function back onto the output:

$$
\begin{aligned}
z_1 &= F_1(x) \\
z_2 &= F_2(z_1) \\
y &= F_3(z_2)
\end{aligned}
\qquad\qquad
\begin{aligned}
z_1 &= F_1(x) + x \\
z_2 &= F_2(z_1) + z_1 \\
y &= F_3(z_2) + z_2
\end{aligned}
$$

The gradients in a network with residual connections are much less sensitive to input values compared to a standard deep network. The effect of the residual connections is to create smoother error function surfaces, because the error surfaces are moderated by a combination of shallow and deep sub-networks.

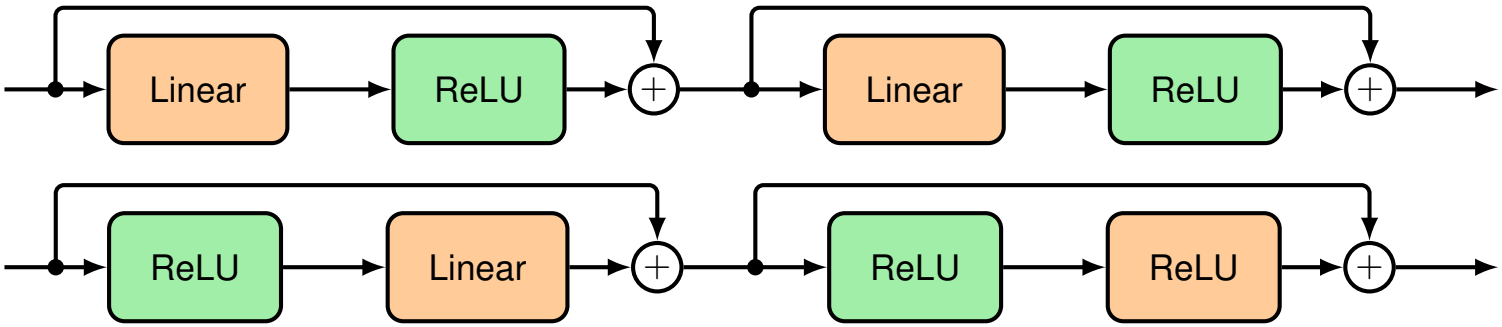
Residual connections

Figure: A residual network and its expanded form



Residual connections

Figure: Two alternative ways to include residual connections into a standard feed-forward network



Model averaging

If we have several different models trained to solve the same problem, we can often improve generalization by averaging the predictions made by the individual models. Consider a regression problem, suppose:

- We have a set of trained models: $y_1(x), \ldots, y_M(x)$.
- The true function that we are trying to predict is given by $h(x)$.
- The output of each of the models can be written as the true value plus an error: $y_m(x) = h(x) + \epsilon_m(x)$.
- We form a committee model given by: $y_{\mathrm{COM}}(x) = \frac{1}{M} \sum_{m=1}^{M} y_m(x)$.

Model averaging

Further, let's suppose that:

- The errors have zero mean: $E(\epsilon_m(x)) = 0$.
- Errors from different models are uncorrelated.

If we calculate the expected error from the committee, we see that:

$$E_{\mathrm{COM}} = E((y_{\mathrm{COM}}(x) - h(x))^2) = E((\frac{1}{M}\sum_{m=1}^{M} y_m(x) - h(x))^2)$$

$$= E((\frac{1}{M}\sum_{m=1}^{M} \epsilon_m(x))^2) = \frac{1}{M^2}\sum_{m=1}^{M} E(\epsilon_m^2(x))$$

$$= \frac{1}{M}(\frac{1}{M}\sum_{m=1}^{M} E((y_m(x) - h(x))^2)) = \frac{1}{M}E_{\mathrm{AV}}$$

where E_{AV} is the average error made by the models acting individually.

Dropout

- The central idea of dropout is to delete nodes from the network, including their connections, at romdom during training.
- Dropout is applied to both hidden nodes and input nodes, but not outputs.
- It can be implemented by defining a mask vector $R_i \in \{0, 1\}$ which multiplies the activation of the non-output node i, whose values are set to 1 with probability ρ.
- $\rho = 0.5$ seems to work well for the hidden nodes, whereas for the inputs a value of $\rho = 0.8$ is typically used.

Dropout

Once training is complete, predictions can in principle be made by:

$$p(y|x) = \sum_{R} p(R)p(y|x, R)$$

where the sum is over the exponentially large space of masks, and $p(y|x, R)$ is the predictive distribution from the network with mask R.

- This summation is intractable, and in practice, as few as 10 or 20 masks can be sufficient to obtain good results. This procedure is known as Monte Carlo dropout.
- A simpler approach is to make predictions using the trained network with no nodes masked out. The weights in the network need to be re-scaled to compensate for the fact that in training a proportion of the nodes would be missing.