1 Notes on Deep Learning

By ***Weikai Mao***

2 Contents

Notes on Deep Learning

- Contents
- Units
  - Single Unit
  - Activation functions
    - Sigmoid, Tanh
    - ReLU, Leaky ReLU
    - Maxout
- Representational Power
- Projection Pursuit Regression
- Feed-Forward Neural Networks
- Training Neural Networks
  - Loss Function
  - Regularization
    - L1 regularization
    - L2 regularization
    - Max Norm Constraints
    - Dropout
  - Optimization
    - Initialization
    - Backpropagation
  - Hyperparameters Tuning
    - Number of Neurons
    - Use a Single Validation Set
    - Hyperparameter Ranges
    - Prefer Random Search
  - Model Ensembles
  - Checks
    - Checks Before Training
    - Gradient Checks
    - Tracking While Training
  - Summary

- Convolutional Neural Networks
  - Architecture Overview
  - CNN Layers
    - Example
    - Convolutional Layer
    - Pooling Layer
  - Layer Patterns
- Recurrent Neural Networks
  - Simple RNNs
  - Stacked and Bidirectional RNNs
    - Stacked RNNs
    - Bidirectional RNNs
- Other Tips
  - Regression and Classification
- References

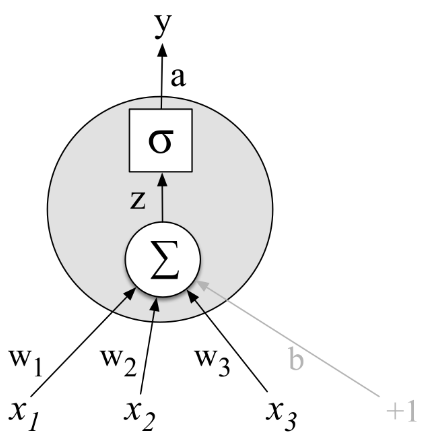
---

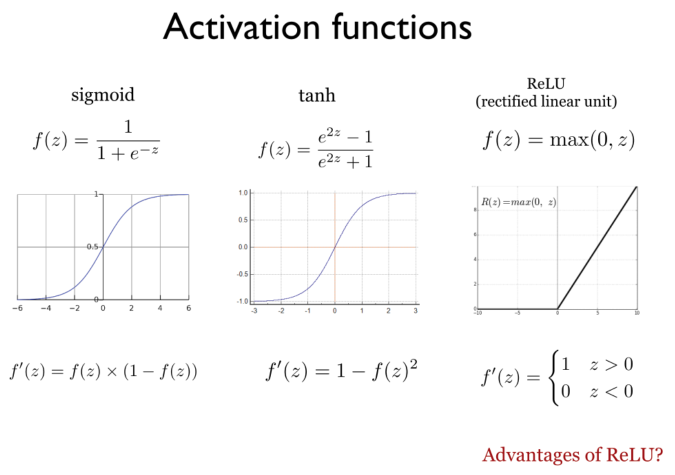
2 Units

3 Single Unit

Taking a weighted sum of its inputs and then applying a activation function. $\sigma(z) = \sigma(w^T x + b)$, where the vector w is called **weight**, the scalar b is called **bias**, the function $\sigma(\cdot)$ is called activation function.

For linear regression, $\sigma(z) = z$. For logistic regression, $\sigma(z) = \frac{1}{1+e^{-z}}$.





3 Activation functions

Activation functions are usually non-linear. If they are all linear, then we simply get the linear combination of the inputs, and that is the same as linear regression.

Sigmoid, Tanh

$$\text{Sigmoid: } \sigma(z) = \frac{1}{1+e^{-z}}; \text{ Tanh: } \sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}.$$

A major drawback of sigmoid and tanh is the **vanishing gradient** problem. When the neuron's activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. Recall that during backpropagation, this (local) gradient will be multiplied to the gradient of this gate's output for the whole objective. Therefore, if the local gradient is very small, it will effectively "kill" the gradient and almost no signal will flow through the neuron to its weights and recursively to its data.

Another drawback of sigmoid is that its outputs are **not zero-centered**. This is not a problem of tanh, and that's why we should use tanh rather than sigmoid. This is a drawback since neurons in later layers of processing in a Neural Network would be receiving data that is not zero-centered. This could introduce undesirable zig-zagging dynamics in the gradient updates for the weights. It has less severe consequences compared to the saturated activation problem above.

ReLU, Leaky ReLU

$$\text{ReLU: } \sigma(z) = \max(z, 0);$$
$$\text{Leaky ReLU: } \sigma(z) = \max(z, \alpha z) = \begin{cases} \alpha z & \text{for } z < 0. \\ z & \text{for } z \geq 0. \end{cases}$$

where α is a small positive number. e.g. 0.01.

Pros of ReLU:

- It greatly **accelerate the convergence** of stochastic gradient descent compared to the sigmoid/tanh functions. (e.g. a factor of 6 in Krizhevsky et al.) It is due to its linear, non-saturating form.
- Computation is easier.

Con of ReLU: ReLU units can be **fragile** during training and can "die". For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. However, with a lower learning rate this is less frequently an issue.

Leaky ReLUs are one attempt to fix the "dying ReLU" problem. Instead of the function being zero when $x < 0$, a leaky ReLU will instead have a small negative slope (of -0.01, or so). However, it not always have benefits.

Maxout

$$\text{Maxout: } \sigma(z^{[1]}, \dots, z^{[m]}) = \max({w^{[1]}}^T x + b^{[1]}, \dots, {w^{[m]}}^T x + b^{[m]})$$

Maxout neuron (introduced recently by Goodfellow et al.) generalizes the ReLU and its leaky version. Notice that both ReLU and Leaky ReLU are a special case of this form (for example, for ReLU we have $\max(w^T x + b, 0^T x + 0)$). The Maxout neuron therefore enjoys all the benefits of a ReLU unit (linear regime of operation, no saturation) and does not have its drawbacks (dying ReLU). However, compared to ReLU, it have m times of the number of parameters for every single neuron, leading to a high total number of parameters.

What neuron type should I use? Use the ReLU non-linearity, be careful with your learning rates and possibly monitor the fraction of "dead" units in a network. If this concerns you, give Leaky ReLU or Maxout a try. Never use sigmoid. Try tanh, but expect it to work worse than ReLU/Maxout.

2 Representational Power

Kolmogorov Arnold Representation theorem states that any continuous function f can be exactly represented by (possible infinite) sequence of addition, multiplication and composition with functions that are universal (do not depend on f).

It can be shown (e.g. see *Approximation by Superpositions of Sigmoidal Function* from 1989 (pdf), or this intuitive explanation from Michael Nielsen) that given any continuous function $f(x)$ and some $\epsilon > 0$, there exists a Neural Network $\hat{f}(x)$ with one hidden layer (with a reasonable choice of non-linearity, e.g. sigmoid) such that $\forall x, \mid f(x) - \hat{f}(x) \mid < \epsilon$. In other words, the one hidden layer neural network can approximate any continuous function.

Despite the fact that their representational power is theoretically equal, in practice, as we increase the size and number of layers in a Neural Network, the capacity of the network increases. That is, the space of representable functions grows since the neurons can collaborate to express many different functions.

2 Projection Pursuit Regression

The projection pursuit regression (PPR) model

$$f(x_i) = \sum_{m=1}^{M} g_m(w_m^T x_i)$$

This is a linear additive model, but in the "derived" features $w_m^T x_i$, rather than the input variables.

The function g_m called ridge function in $\mathbb{R}^p$ and is a non-linear function. The $p \times 1$ vector w_m is called the inner-layer or hidden layer weights vector. The scalar M is is called the width (the number of hidden units). The scalar m is the index of hidden unit. The scalar variable $w_m^T x_i$ is the projection of x_i onto the unit vector w_m, and we seek w_m so that the model fits well, hence the name "projection pursuit".

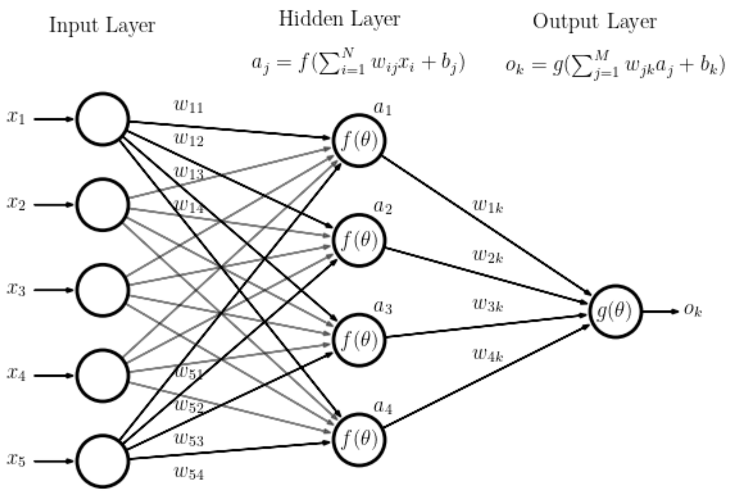
If M is taken arbitrarily large, for appropriate choices of g_m, f can approximate any continuous function to any desired level of accuracy. Note: when $M = 1$, i.e. $f_1(X) = g_1(w_m^T X)$, which is called single hidden layer.

For classification, we can use log loss function. For regression, we can use squared error loss function $\sum_{i=1}^{N} L(y_i, \hat{y}_i) = \sum_{i=1}^{N} \left[y_i - \sum_{m=1}^{M} g_m(w_m^T x_i) \right]^2$.

We seek the approximate minimizers g_m's and w_m's of the error function. Given g_m's, we want to minimize over w_m's. A Gauss-Newton search is convenient for this task. This is a quasi-Newton method.

2 Feed-Forward Neural Networks

A **feedforward network** is a multilayer network in which the units are connected with no cycles; the outputs from units in each layer are passed to units in the next higher layer, and no outputs are passed back to lower layers. Simple feedforward networks have three kinds of nodes: input units, hidden units, and output units. The core of the neural network is the **hidden layer** formed of **hidden units**, each of which is a neural unit. In the standard architecture, each layer is **fully-connected**.



Let's assume there are n_0 hidden units in hidden layer 0 and n_1 units in layer 1. The output of the hidden layer 0 can be expressed by $x_1 = \sigma(Wx_0 + b)$, where the outputs $x_1 \in \mathbb{R}^{n_1}$, the weight matrix $W \in \mathbb{R}^{n_1 \times n_0}$, the inputs $x \in \mathbb{R}^{n_0}$, the bias vector $b \in \mathbb{R}^{n_1}$.

We apply **softmax** function at output layer for K-class classification:

$$\sigma(\mathbf{z})_l = \frac{e^{z_l}}{\sum_{k=1}^{K} e^{z_k}} \text{ for } l = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K.$$

The output layer thus gives the **estimated probabilities** of each output node (corresponds to each class). If the number of classes is too large, it may be helpful to use Hierarchical Softmax.

The activation function will tend to be Projection Pursuit Regression

Kolmogorov Arnold Representation theorem states that any continuous function f can be exactly represented by (possible infinite) sequence of addition, multiplication and composition with functions that are universal (do not depend on f).

The projection pursuit regression (PPR) model

$$f(x_i) = \sum_{m=1}^{M} g_m(w_m^T x_i)$$

This is a linear additive model, but in the "derived" features $w_m^T x_i$, rather than the input variables.

The function g_m called ridge function in $\mathbb{R}^p$ and is a non-linear function. The $p \times 1$ vector w_m is called the inner-layer or hidden layer weights vector. The scalar M is is called the width (the number of hidden units). The scalar m is the index of hidden unit. The scalar variable $w_m^T x_i$ is the projection of x_i onto the unit vector w_m, and we seek w_m so that the model fits well, hence the name "projection pursuit".

If M is taken arbitrarily large, for appropriate choices of g_m, f can approximate any continuous function to any desired level of accuracy. Note: when $M = 1$, i.e. $f_1(X) = g_1(w_m^T X)$, which is called single hidden layer.

For classification, we can use log loss function. For regression, we can use squared error loss function $\sum_{i=1}^N L(y_i, \hat{y}_i) = \sum_{i=1}^N \left[y_i - \sum_{m=1}^M g_m(w_m^T x_i)\right]^2$.

We seek the approximate minimizers g_m's and w_m's of the error function. Given g_m's, we want to minimize over w_m's. A Gauss-Newton search is convenient for this task.

We traditionally don’t count the input layer when numbering layers, but do count the output layer. So by this terminology logistic regression is a 1-layer network.

2 Training Neural Networks

The goal of the training procedure is to learn parameters $W^{[l]}$ and $b^{[l]}$ for each layer l that minimize the loss function plus regularization term.

3 Loss Function

Cross-entropy loss function for K-class classification: $L(y_i, \hat{p}_i) = -\sum_{k=1}^K I(y_i \text{ in class } k) \log(\hat{p}_{ik})$, where $\hat{p}_{ik}$ is the estimated probability of y_i belongs to class k. For binary classification, $L(y_i, \hat{p}_i) = -y_i \log(\hat{p}_i) - (1 - y_i) \log(1 - \hat{p}_i)$, where $y_i \in \{0, 1\}$, $\hat{p}_i$ is the estimated probability of $y_i = 1$. The estimated probabilities $\hat{p}_l$ by softmax function of the class l is $e^{z_l} / \sum_{k=1}^K e^{z_k}$.

3 Regularization

We usually need to regularize the weight parameters, but It is not common to regularize the bias parameters because they do not interact with the data through multiplicative interactions, and therefore do not have the interpretation of controlling the influence of a data dimension on the final objective.

Why we don't control the depth of networks to avoid overfitting?

In practice, it is always better to use the following methods to control overfitting instead of controlling the depth. In terms of optimization of objective, compared to shallow NNs, the deeper NNs contain significantly more local minima, but these minima turn out to be much better than that of shallow NNs in terms of their actual loss. Since Neural Networks are non-convex, it is hard to study these properties mathematically.

Additionally, the final loss of a shallower network can have a higher variance in practice, it may due to the number of minimas is small. Sometimes we get a good minima, and sometimes we get a very bad one, and it relies on the random initialization. However, for a deep network, all solutions are about equally as good, and rely less on the luck of random initialization.

In conclusion, you should use as big of a neural network as your computational budget allows, and use other regularization techniques to control overfitting.

L1 regularization

L1 regularization is another relatively common form of regularization, where for each weight w we add the term $\lambda \mid w|$ to the objective. It is possible to combine the L1 regularization with the L2 regularization: $\lambda_1 \mid w \mid + \lambda_2 w^2$ (this is called Elastic net regularization).

The L1 regularization has the intriguing property that it leads the weight vectors to become sparse during optimization (i.e. very close to exactly zero). In other words, neurons with L1 regularization end up using only a sparse subset of their most important inputs and become nearly invariant to the "noisy" inputs.

L2 regularization

L2 regularization is perhaps the most common form of regularization. It can be implemented by penalizing the squared magnitude of all parameters directly in the objective. That is, for every weight w in the network, we add the term $\frac{1}{2}\lambda w^2$ to the objective, where λ is the regularization strength. (It is common to see the factor of $1/2$ in front because then the gradient of this term with respect to the parameter w is simply λw instead of $2\lambda w$.)

The L2 regularization has the intuitive interpretation of heavily penalizing peaky weight vectors and preferring diffuse weight vectors. Due to multiplicative interactions between weights and inputs this has the appealing property of encouraging the network to use all of its inputs a little rather than some of its inputs a lot. Lastly, notice that during gradient descent parameter update, using the L2 regularization ultimately means that every weight is decayed linearly: `W += -lambda * W` towards zero.

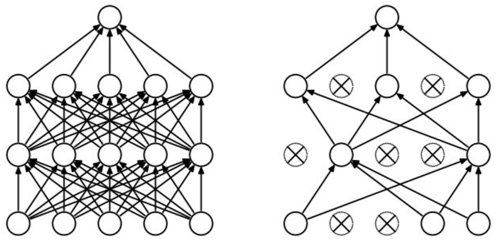
In comparison, final weight vectors from L2 regularization are usually diffuse, small numbers. In practice, if you are not concerned with explicit feature selection, L2 regularization can be expected to give superior performance over L1.

Max Norm Constraints

Another form of regularization is to enforce an absolute upper bound on the magnitude of the weight vector for every neuron and use projected gradient descent to enforce the constraint. In practice, this corresponds to performing the parameter update as normal, and then enforcing the constraint by clamping the weight vector w of every neuron to satisfy $\|w\|_2 < c$. Typical values of c are on orders of 3 or 4. Some people report improvements when using this form of regularization. One of its appealing properties is that network cannot "explode" even when the learning rates are set too high because the updates are always bounded.

Dropout

Dropout is an extremely effective, simple and recently introduced regularization technique by Srivastava et al. in Dropout: A Simple Way to Prevent Neural Networks from Overfitting (pdf) that complements the other methods (L1, L2, maxnorm). While training, dropout is implemented by only keeping a neuron active with some probability pp (a hyperparameter), or setting it to zero otherwise.


(a) Standard Neural Net (b) After applying dropout.

3 Optimization

4 Initialization

If the weights in a network start too small/large, then the signal shrinks/grows as it passes through each layer until it is too tiny/massive to be useful, because it may lead to vanishing or exploding gradients.

The appropriate initialization should have the following rules of thumb:

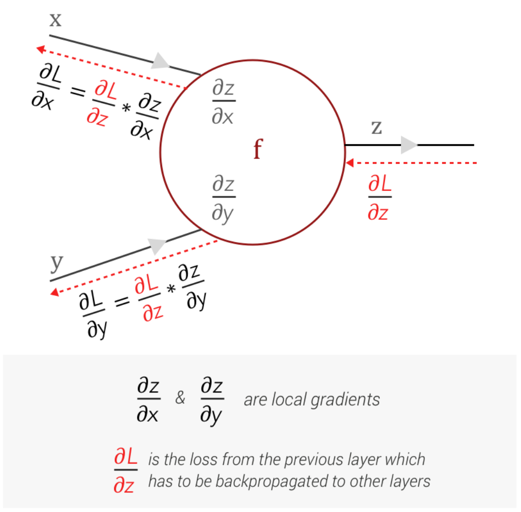
1. The mean of the activations should be zero.
2. The variance of the activations should stay the same across every layer.

Xavier initialization: For every layer l: $W^{[l]} \sim N(0, 1/n^{[l-1]}),\ b^{[l]} = 0$.

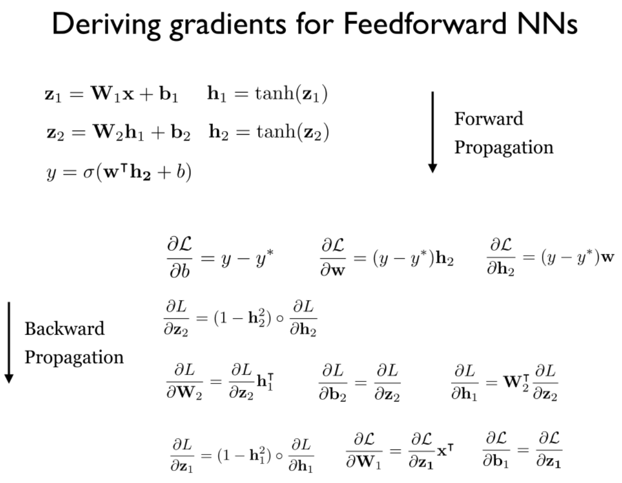
4 Backpropagation

We use GD or SGD to optimize the objective function $Obj(w)$. For weight, we have $w_t = w_{t-1} - \alpha \frac{\partial\ Obj(w_{t-1})}{\partial\ w_{t-1}}$, where α is the step-size (learning rate).

The solution to computing the gradient is an algorithm called error **backpropagation**, which use the chain rules in calculus. (Note: the regularization term is not considered in the pictures below.)



x
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} * \frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial x}$
f
$\frac{\partial z}{\partial y}$
z
$\frac{\partial L}{\partial z}$
y
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} * \frac{\partial z}{\partial y}$
$\frac{\partial z}{\partial x}$ & $\frac{\partial z}{\partial y}$ are local gradients
$\frac{\partial L}{\partial z}$ is the loss from the previous layer which has to be backpropagated to other layers


Deriving gradients for Feedforward NNs

$$\mathbf{z}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1 \quad \mathbf{h}_1 = \tanh(\mathbf{z}_1)$$
$$\mathbf{z}_2 = \mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2 \quad \mathbf{h}_2 = \tanh(\mathbf{z}_2)$$
$$y = \sigma(\mathbf{w}^\intercal\mathbf{h_2} + b)$$

Forward Propagation

$$\frac{\partial \mathcal{L}}{\partial b} = y - y^* \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{w}} = (y - y^*)\mathbf{h}_2 \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{h}_2} = (y - y^*)\mathbf{w}$$
$$\frac{\partial L}{\partial \mathbf{z}_2} = (1 - \mathbf{h}_2^2) \circ \frac{\partial L}{\partial \mathbf{h}_2}$$
$$\frac{\partial L}{\partial \mathbf{W}_2} = \frac{\partial L}{\partial \mathbf{z}_2}\mathbf{h}_1^\intercal \qquad \frac{\partial L}{\partial \mathbf{b}_2} = \frac{\partial L}{\partial \mathbf{z}_2} \qquad \frac{\partial L}{\partial \mathbf{h}_1} = \mathbf{W}_2^\intercal \frac{\partial L}{\partial \mathbf{z}_2}$$
$$\frac{\partial L}{\partial \mathbf{z}_1} = (1 - \mathbf{h}_1^2) \circ \frac{\partial L}{\partial \mathbf{h}_1} \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{W}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{z_1}}\mathbf{x}^\intercal \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{b}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{z_1}}$$

Backward Propagation

The Backpropagation Algorithm: The derivatives of Loss with respect to any of the weights can be computed, layer by layer, as follows:

- At $t = K$, compute
$$\Delta_j^K = \frac{\partial L}{\partial \text{out}_j^K}, \tag{23}$$
for each output node j, based on the specific loss function.
- For $t < K$ (in decreasing order), compute
$$\Delta_j^t = \frac{\partial L}{\partial \text{out}_j^t} = \sum_k \Delta_k^{t+1} \sigma' \left(\underline{w}^t(k) . \underline{\text{out}}^t \right) w_{j,k}^t. \tag{24}$$
- Any weight can then be updated according to
$$(\text{new } w_{i,j}^t) = w_{i,j}^t - \alpha \frac{\partial L}{\partial w_{i,j}^t} = w_{i,j}^t - \alpha \Delta_j^{t+1} \sigma' \left(\underline{w}^t(j) . \underline{\text{out}}^t \right) \text{out}_i^t. \tag{25}$$

where t is the index of the layer.

3 Hyperparameters Tuning

The most common hyperparameters in Neural Networks include:

- the number of neurons (width)
- the number of layers (depth)
- the initial learning rate
- learning rate decay schedule (such as the decay constant)
- regularization strength (L2 penalty, dropout strength)

4 Number of Neurons

Here are 3 rule-of-thumb methods for determining an acceptable number of neurons to use in the hidden layers [Reference]. The number of hidden neurons should be:

- between the size of the input layer and the size of the output layer.
- 2/3 the size of the input layer, plus the size of the output layer.
- less than twice the size of the input layer.

4 Use a Single Validation Set

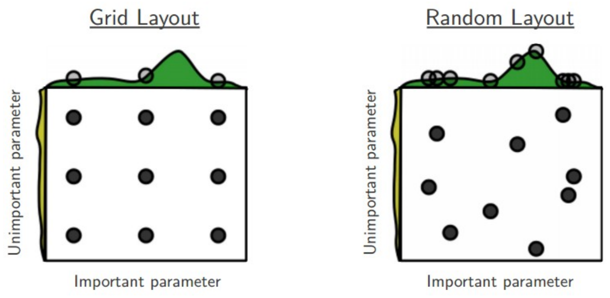
In practice, we prefer one validation fold to cross-validation. In most cases a single validation set of respectable size substantially simplifies the code base, without the need for cross-validation with multiple folds.

4 Hyperparameter Ranges

Search for learning rate and regularization strength on log scale. For example, a typical sampling of the learning rate would look as follows: `learning_rate = 10 ** uniform(-6, 1)`. That is, we are generating a random number from a uniform distribution, but then raising it to the power of 10.

4 Prefer Random Search

We prefer random search to grid search. As argued by Bergstra and Bengio in Random Search for Hyper-Parameter Optimization, "randomly chosen trials are more efficient for hyper-parameter optimization than trials on a grid". As it turns out, this is also usually easier to implement.



3 Model Ensembles

One approach to improving the performance of NNs is to train multiple independent models, and then average their predictions. As the number of models in the ensemble increases, the performance typically monotonically improves (though with diminishing returns). The improvements are more dramatic with higher model variety in the ensemble.

There are a few approaches to forming an ensemble:

- **Same model, different initializations**. Use cross-validation to determine the best hyperparameters, then train multiple models with the best set of hyperparameters but with different random initialization. The danger with this approach is that the variety is only due to initialization.
- **Top models discovered during cross-validation**. Use cross-validation to determine the best hyperparameters, then pick the top few (e.g. 10) models to form the ensemble. This improves the variety of the ensemble but has the danger of including suboptimal models. In practice, this can be easier to perform since it doesn’t require additional retraining of models after cross-validation
- **Different checkpoints of a single model**. If training is very expensive, some people have had limited success in taking different checkpoints of a single network over time (for example after every epoch) and using those to form an ensemble. Clearly, this suffers from some lack of variety, but can still work reasonably well in practice. The advantage of this approach is that is very cheap.
- **Running average of parameters during training**. Related to the last point, a cheap way of almost always getting an extra percent or two of performance is to maintain a second copy of the network’s weights in memory that maintains an exponentially decaying sum of previous weights during training. This way you’re averaging the state of the network over last several iterations. You will find that this “smoothed” version of the weights over last few steps almost always achieves better validation error. The rough intuition to have in mind is that the objective is bowl-shaped and your network is jumping around the mode, so the average has a higher chance of being somewhere nearer the mode.

One disadvantage of model ensembles is that they take longer to evaluate on test example. An interested reader may find the recent work from Geoff Hinton on “Dark Knowledge” inspiring, where the idea is to “distill” a good ensemble back to a single model by incorporating the ensemble log likelihoods into a modified objective.

3 Checks

4 Checks Before Training

Here are a few sanity checks you might consider running before you plunge into expensive optimization:

- Make sure you're getting the loss you expect when you initialize with small parameters. It's best to first check the data loss alone (so set regularization strength to zero).
- Increasing the regularization strength should increase the training loss.
- Overfit a tiny subset of data. Lastly and most importantly, before training on the full dataset try to train on a tiny portion of your data and make sure you can achieve zero cost. For this experiment it's also best to set regularization to zero.

4 Gradient Checks

- **Use Centered Formula**

Denote the objective function as $f(\theta)$, where θ is a w or b (note that it is not the parameter we need to tune). Backprop as an algorithm has a lot of details and can be a little bit tricky to implement, so there are many ways to have subtle bugs in backprop. Gradient checking is kind of debugging your backprop algorithm. We compute the **analytic gradient** $f'_a(\theta)$ by backprop and compare it with the **numerical gradient** $f'_n(\theta)$ which is evaluted by centered formula as follows:

$$f'_n(\theta) = \frac{df(\theta)}{d\theta} \approx \frac{f(\theta+\Delta) - f(\theta-\Delta)}{2\Delta}.$$

We use the previous formula rather than $\frac{L(\theta+\Delta)-L(\theta)}{\Delta}$, because, by Taylor expansion, the first formula has an error on order of $O(\Delta^2)$, while the second one has $O(\Delta)$.

- **Be Care of Δ**

It is not the case that the smaller Δ is better, because when Δ is much smaller, you may start running into numerical precision problems. In practice, **Δ is between** 10^{-6} **and** 10^{-4}.

- **Use Relative Error for Comparison**

How to compare the numerical gradient f'_n and analytic gradient f'_a? We use the relative error

$$\frac{|f'_a - f'_n|}{\max(|f'_a|, |f'_n|)}.$$

which considers their ratio of the differences to the ratio of the absolute values of both gradients.

In practice:

- relative error > 1e-2 usually means the gradient is probably wrong
- 1e-2 > relative error > 1e-4 should make you feel uncomfortable
- 1e-4 > relative error is usually okay for objectives with kinks. But if there are no kinks (e.g. use of tanh nonlinearities and softmax), then 1e-4 is too high.

- 1e-7 and less you should be happy.

The deeper the network, the higher the relative errors will be. So if you are gradient checking the input data for a 10-layer network, a relative error of 1e-2 might be okay because the errors build up on the way.

- **Problem of Kinks**

One source of inaccuracy to be aware of during gradient checking is the problem of kinks. Kinks refer to non-differentiable parts of an objective function, introduced by functions such as ReLU ($\max(0, z)$), or the SVM loss, Maxout neurons, etc.

Consider gradient checking the ReLU function at $z = 10^{-6}$. Since $x < 0$, the analytic gradient at this point is exactly zero. However, the numerical gradient would suddenly compute a non-zero gradient because $L(w + \Delta)$ might cross over the kink (e.g. if $\Delta > 10^{-6}$) and introduce a non-zero contribution.

One fix to the above problem of kinks is to use fewer datapoints, since loss functions that contain kinks (e.g. due to use of ReLUs or margin losses etc.) will have fewer kinks with fewer datapoints, so it is less likely for you to cross one when you perform the finite different approximation. Using very few datapoints also makes your gradient check faster.

- **Be Care of Regularization Term**

The regularization loss may overwhelm the data loss, in which case the gradients will be primarily coming from the regularization term. This can mask an incorrect implementation of the data loss gradient. Therefore, it is recommended to turn off regularization and check the data loss alone first, and then the regularization term second and independently.

- **Force a Seed for Random Effects**

We need to eliminate any non-deterministic effects (e.g. dropout, random data augmentations, etc.) in the network when performing gradient check. Thus, we should force a particular random seed before evaluating both $f(\theta + \Delta)$ and $f(\theta - \Delta)$, and when evaluating the analytic gradient.

- **Check Only Few Parameters**

In practice the gradients can have sizes of million parameters. In these cases it is only practical to check some of the dimensions of the gradient in one check and assume that the others are correct. We should also make sure we have checked all the parameters after all checks have been done.

4 Tracking While Training

We should track training loss, validation loss and the ratio of the update magnitudes of the parameters.

A rough heuristic is that the ratio should be somewhere around 1e-3. If it is lower than this then the learning rate might be too low. If it is higher then the learning rate is likely too high. Here is a specific example:

```python
# assume weight vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print(update_scale / param_scale) # want ~1e-3
```

3 Summary

To train a Neural Network:

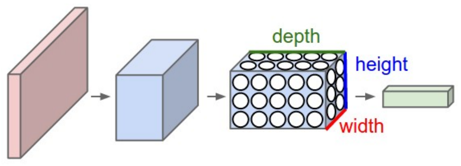
- Gradient check your implementation with a small batch of data and be aware of the pitfalls.
- As a sanity check, make sure your initial loss is reasonable, and that you can achieve 100% training accuracy on a very small portion of the data
- During training, monitor the loss, the training/validation accuracy, and if you're feeling fancier, the magnitude of updates in relation to parameter values (it should be ~1e-3), and when dealing with ConvNets, the first-layer weights.
- The two recommended updates to use are either SGD+Nesterov Momentum or Adam.
- Decay your learning rate over the period of the training. For example, halve the learning rate after a fixed number of epochs, or whenever the validation accuracy tops off.
- Search for good hyperparameters with random search (not grid search). Stage your search from coarse (wide hyperparameter ranges, training only for 1-5 epochs), to fine (narrower rangers, training for many more epochs).
- Form model ensembles for extra performance.

2 Convolutional Neural Networks

3 Architecture Overview

ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.

The full connectivity is wasteful and the huge number of parameters would quickly lead to overfitting. A ConvNet arranges its neurons in three dimensions (width, height, depth), and the neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner.



A ConvNet is made up of Layers. **Every layer transforms an input 3D volume to an output 3D volume with some differentiable function** that may or may not have parameters. As visualized in the left picture above, a layer in ConvNet arranges its neurons in three dimensions (width, height, depth). In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

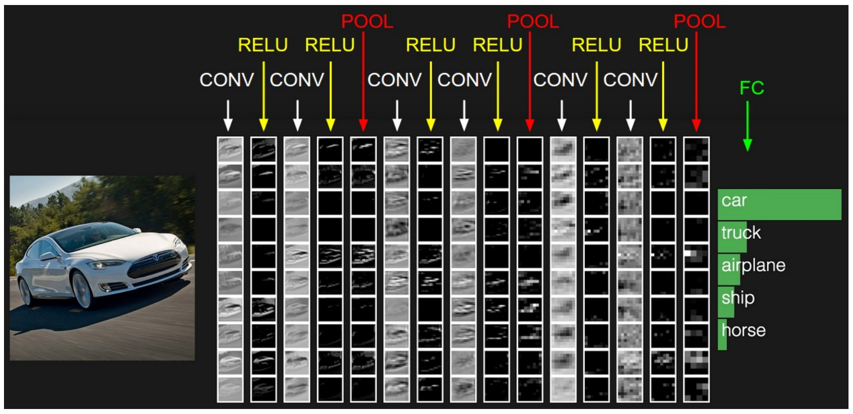
3 CNN Layers

We use three main types of layers to build ConvNet architectures: **Convolutional Layer**, **Pooling Layer**, and **Fully-Connected Layer** (exactly as seen in regular Neural Networks). We will stack these layers to form a full ConvNet architecture.

4 Example

Example Architecture: In CIFAR-10, images are only of size 32x32x3 (32 wide, 32 high, 3 color channels). A simple ConvNet for CIFAR-10 classification could have the architecture [INPUT - CONV - RELU - POOL - FC]. In more detail:

- INPUT [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.
- CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as [32x32x12] if we decided to use 12 filters.
- RELU layer will apply an elementwise activation function, such as the $\max(0, z)$ thresholding at zero. This leaves the size of the volume unchanged ([32x32x12]).
- POOL layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x12].
- FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.



The picture above shows the activations of an example ConvNet architecture. The initial volume stores the raw image pixels (left) and the last volume stores the class scores (right). Each volume of activations along the processing path is shown as a column. Since it's difficult to visualize 3D volumes, we lay out each volume's slices in rows. The last layer volume holds the scores for each class, but here we only visualize the sorted top 5 scores

In summary:

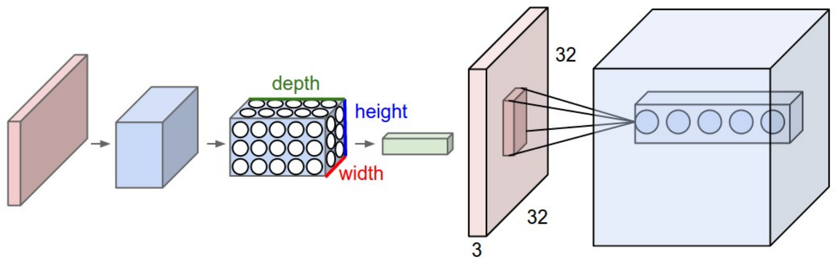
- A ConvNet architecture is in the simplest case a list of Layers that transform the image volume into an output volume (e.g. holding the class scores)
- There are a few distinct types of Layers (e.g. CONV/FC/RELU/POOL are by far the most popular)
- Each Layer accepts an input 3D volume and transforms it to an output 3D volume through a differentiable function
- Each Layer may or may not have parameters (e.g. CONV/FC do, RELU/POOL don't)
- Each Layer may or may not have additional hyperparameters (e.g. CONV/FC/POOL do, RELU doesn't)

4 Convolutional Layer

The Conv layer is the core building block of a Convolutional Network that does most of the computational heavy lifting.

- **Local Connectivity**

We connect each neuron in the convolutional layer (blue) to only a local region of the input volume (red). The spatial extent of this connectivity is a hyperparameter called the **receptive field** of the neuron (equivalently this is the filter size).



As shown in the picture above, suppose that the input volume (red) has size [32x32x3], (e.g. an RGB CIFAR-10 image). If the receptive field (or the filter size) is 5x5, then each neuron in the Conv Layer (blue) will have weights to a [5x5x3] region in the input volume, for a total of 553 = 75 weights (and +1 bias parameter). Each neuron in the convolutional layer is connected only to a local region in the input volume spatially, but to the full depth (i.e. all color channels). Note, there are multiple neurons (5 in this example) along the depth, all looking at the same region in the input.

- **Spatial Arrangement**

How many neurons there are in the convolutional layer? How they are arranged? They are controlled by three hyperparameters: the **depth, stride** and **zero-padding**.

1. First, the **depth** of the output volume is a hyperparameter (it is 5 as shown in the picture above): it corresponds to the number of filters we would like to use, each learning to look for something different

in the input. For example, if the first Convolutional Layer takes as input the raw image, then different neurons along the depth dimension may activate in presence of various oriented edges, or blobs of color. We will refer to a set of neurons that are all looking at the same region of the input as a depth column (or fibre).
2. Second, we must specify the **stride** with which we slide the filter. When the stride is 1 then we move the filters one pixel at a time. When the stride is 2 (or uncommonly 3 or more, though this is rare in practice) then the filters jump 2 pixels at a time as we slide them around. This will produce smaller output volumes spatially.
3. Third, the size of **zero-padding**. We can pad the input volume with zeros around the border to control the spatial size of the output volumes (most commonly we use it to make the output volume has the same width and height with the input volume).

Overall, there are four parameters: the receptive field size of the Conv Layer neurons (F); the depth of the layer (or the number of filters) (K); the stride (S); and the amount of zero padding used (P) on the border.

Denote the input volume size as V. The number of the neurons in the convolutional layer is $(V - F + 2P)/S + 1$. We usually set $P = \frac{1}{2}(VS - V - S + F)$ to ensure that the input volume and output volume have the same size spatially.

- **Parameter Sharing**

Parameter sharing scheme is used to reduce the number of parameters (weights and biases). The assumption is that if one feature is useful to compute at some spatial position (x_1, y_1), then it should also be useful to compute at a different position (x_2, y_2). In other words, denoting a single 2-dimensional slice of depth as a **depth slice** (e.g. a volume of size [55x55x96] has 96 depth slices, each of size [55x55]), we are going to constrain the neurons in each depth slice to use the same weights and bias. Each depth slice is computed as a **convolution** of the neuron's weights with the input volume (Hence the name: Convolutional Layer).

Note that sometimes the parameter sharing assumption may not make sense. One practical example is when the input are faces that have been centered in the image. You might expect that different eye-specific or hair-specific features could (and should) be learned in different spatial locations. In that case it is common to relax the parameter sharing scheme, and instead simply call the layer a **Locally-Connected Layer**.

- **Summary**

To summarize, the Conv Layer:

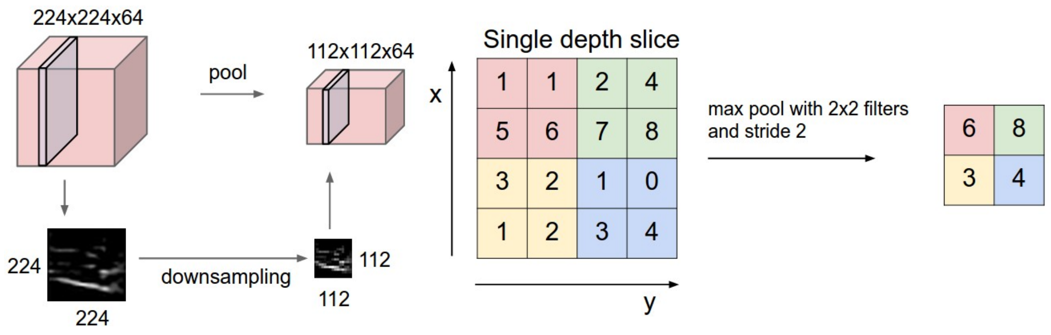
- Accepts a volume of size $W_1 \times H_1 \times D_1$.
- Requires four hyperparameters:
  - depth of the layer (Number of filters) K;
  - receptive field size F;
  - stride S;
  - amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$, where:
  - $W_2 = (W_1 - F + 2P)/S + 1$;
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry);
  - $D_2 = K$.

- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

In practice, a common setting of the hyperparameters is $F = 3, S = 1, P = 1$. However, there are common conventions and rules of thumb that motivate these hyperparameters. See the ConvNet architectures section below.

4 Pooling Layer

It is common to periodically insert a Pooling layer in-between successive Conv layers. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the operations like max, average or L2-norm. In practice, max pooling operation normally works better.



The pooling layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$.
- Requires two hyperparameters:
  - receptive field size F;
  - stride S.
- Produces a volume of size $W_2 \times H_2 \times D_2$, where:
  - $W_2 = (W_1 - F)/S + 1$;
  - $H_2 = (H_1 - F)/S + 1$;
  - $D_2 = D_1$.
- Introduces zero parameters since it computes a fixed function of the input
- For Pooling layers, it is not common to pad the input using zero-padding.

Pooling sizes with larger receptive fields are too destructive. There are two commonly seen variations of the max pooling layer:

- $F = 3, S = 2$ (also called overlapping pooling);
- $F = 2, S = 2$ (more comonly).

Getting rid of pooling. Many people dislike the pooling operation and think that we can get away without it. For example, Striving for Simplicity: The All Convolutional Net proposes to discard the pooling layer in favor of architecture that only consists of repeated CONV layers. To reduce the size of the representation they suggest using larger stride in CONV layer once in a while. Discarding pooling layers has also been found to be important in training good generative models, such as variational autoencoders (VAEs) or generative adversarial networks (GANs). It seems likely that future architectures will feature very few to no pooling layers.

3 Layer Patterns

The most common form of a ConvNet architecture stacks a few CONV-RELU layers, follows them with POOL layers, and repeats this pattern until the image has been merged spatially to a small size, then follows with fully-connected layers. The last fully-connected layer holds the output, such as the class scores.

```
INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC
```

where the `*` indicates repetition, and the `POOL?` indicates an optional pooling layer. Moreover, `N >= 0` (and usually `N <= 3`), `M >= 0`, `K >= 0` (and usually `K < 3`).

2 Recurrent Neural Networks

A recurrent neural network (RNN) is any network that contains a cycle within its network connections. That is, any network where the value of a unit is directly, or indirectly, dependent on earlier outputs as an input.

3 Simple RNNs

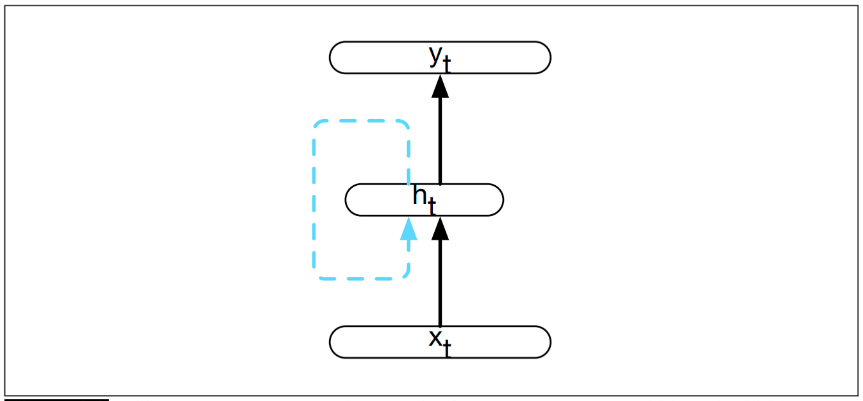


Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous time step.

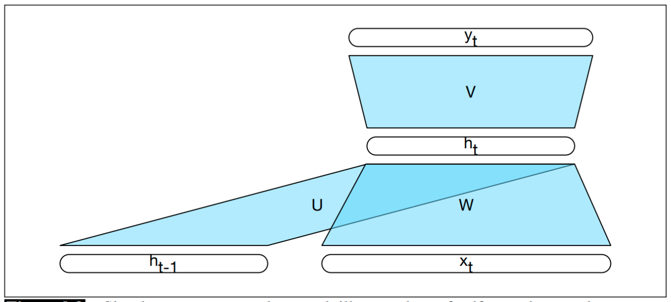


Figure 9.3 Simple recurrent neural network illustrated as a feedforward network.

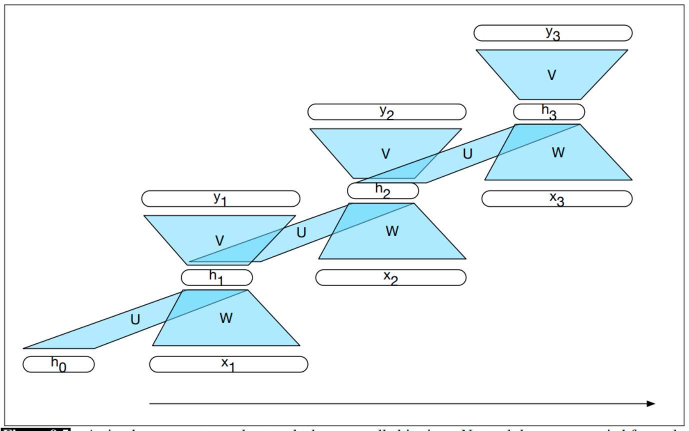


Figure 9.5 A simple recurrent neural network shown unrolled in time. Network layers are copied for each time step, while the weights *U*, *V* and *W* are shared in common across all time steps.

```
function FORWARD_RNN(x, network) returns output sequence y:
    h0 = 0
    for t = 1 to LENGTH(x) do:
        ht = g(U * ht + W * xt)
        yi = f(V * hi)
    return y
```

3 Stacked and Bidirectional RNNs

4 Stacked RNNs

In stacked RNNs, we use the entire sequence of outputs from one RNN as an input sequence to another one.

Stacked RNNs can usually outperform single-layer networks.

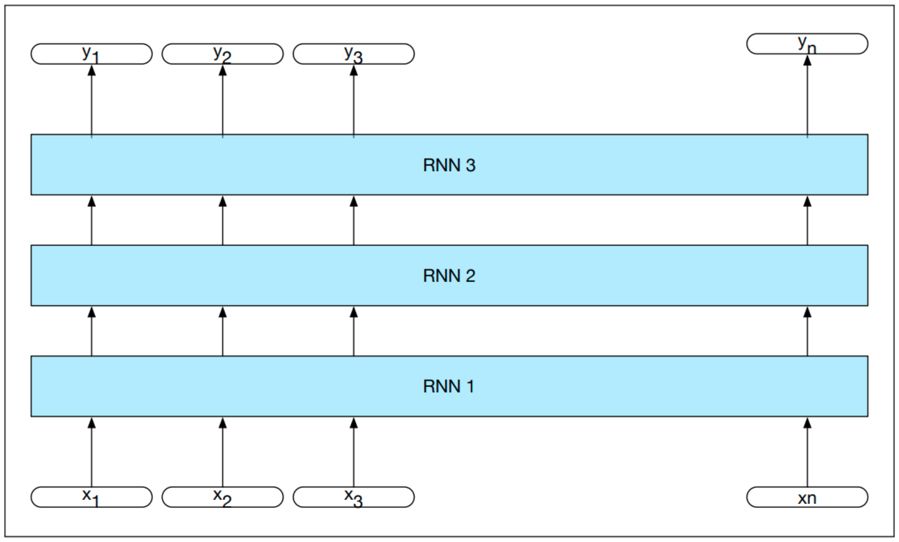


Figure 9.10 Stacked recurrent networks. The output of a lower level serves as the input to higher levels with the output of the last network serving as the final output.

4 Bidirectional RNNs

We can train an bacward RNN on an input sequence in reverse. Combining the forward and backward networks results in a **bidirectional RNN**. We can use concatenation or simply element-wise addition or multiplication to combine the results of the two RNNs.

The bidirectional RNN for sequence labeling:

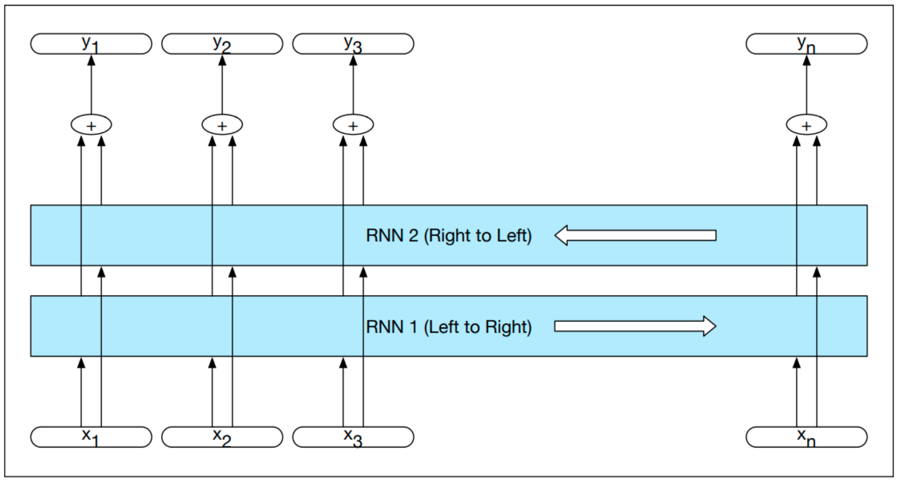


Figure 9.11 A bidirectional RNN. Separate models are trained in the forward and backward directions with the output of each model at each time point concatenated to represent the state of affairs at that point in time. The box wrapped around the forward and backward network emphasizes the modular nature of this architecture.

The bidirectional RNN for sequence classification:

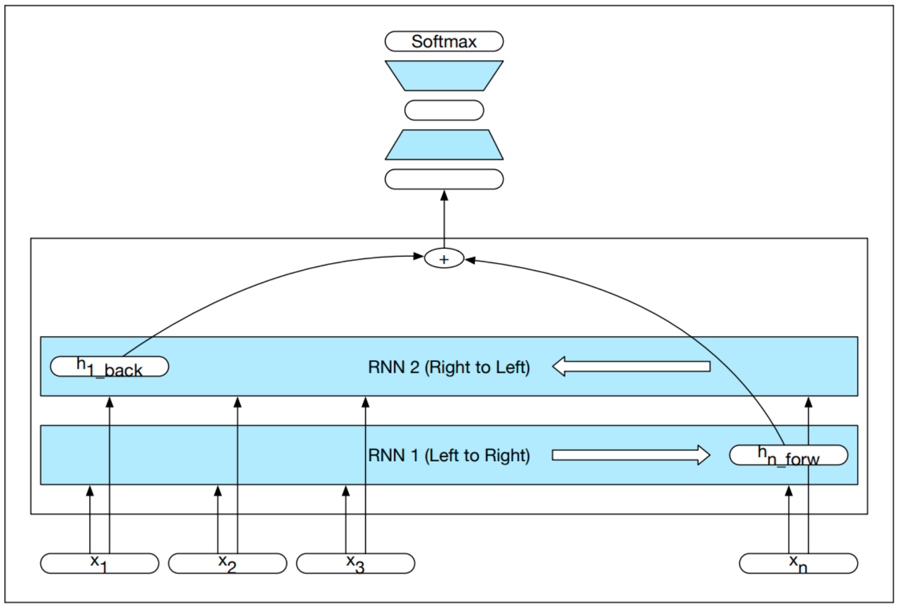


Figure 9.12 A bidirectional RNN for sequence classification. The final hidden units from the forward and backward passes are combined to represent the entire sequence. This combined representation serves as input to the subsequent classifier.

Bidirectional RNNs have also proven to be quite effective for sequence classification.

2 Other Tips

3 Regression and Classification

As for neural networks, the L2 loss (MSE) for regression problem is much harder to optimize than a more stable loss such as Softmax for classification problem. Intuitively, it requires a very specific property from the network to output exactly one correct value for each input (and its augmentations). The L2 loss is less robust because outliers can introduce huge gradients.

When faced with a regression task, first consider if it is absolutely necessary. Instead, have a strong preference to discretizing your outputs to bins and perform classification over them whenever possible. For example, if you are predicting star rating for a product, it might work much better to use 5 independent classifiers for ratings of 1-5 stars instead of a regression loss.

If you're certain that classification is not appropriate, use the L2 but be careful: For example, the L2 is more fragile and applying dropout in the network (especially in the layer right before the L2 loss) is not a great idea.

2 References

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

Jurafsky, Daniel and James H. Martin. *Speech and Language Processing (3rd ed.)*. 2019.

Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*. Vol. 1. No. 10. New York: Springer series in statistics, 2001.

CS231n Convolutional Neural Networks for Visual Recognition